ARMY RESEARCH LABORATORY

Working with Evenly Spaced Rectangular Surface Grids Using C++

by Robert J Yager

ARL-TN-0641 **October 2014**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5066

ARL-TN-0641 **October 2014**

Working with Evenly Spaced Rectangular Surface Grids Using C++

Robert J Yager
Weapons and Materials Research Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
October 2014	Final	April 2012–May 2014

4. TITLE AND SUBTITLE
Working with Evenly Spaced Rectangular Surface Grids Using C++

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Robert J Yager

5d. PROJECT NUMBER
AH80

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-WML-A
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TN-0641

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report presents a set of functions, written in C++, that is designed to work with evenly spaced rectangular surface grids. Grids of this type have a variety of applications, including representing terrain features and storing spatial probability information. Relative to other methods of storing surface information, they can be advantageous when calculation time is critical.

15. SUBJECT TERMS
bilinear, interpolation, intersection, gradient, C++, terrain

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Robert J Yager
Unclassified	Unclassified	Unclassified	UU	44	19b. TELEPHONE NUMBER (*Include area code*) (410) 278-6689

Contents

List of Figures

Acknowledgments

I would like to thank Mary Arthur of the US Army Research Laboratory's Weapons and Materials Research Directorate. Mary provided technical and editorial recommendations that improved the quality of this report.

1. Introduction

This report presents a set of functions, written in C++, that is designed to work with evenly spaced rectangular surface grids. Grids of this type have a variety of applications, including representing terrain features and storing spatial probability information. Relative to other methods of storing surface information, they can be advantageous when calculation time is critical.

The functions that are described in this report have been grouped into the yBilinear namespace, which is summarized at the end of this report. Functions for calculating surface interpolations, line-surface intersections, and surface gradients are included. The functions are based on bilinear interpolations and have been templated to allow for a variety of pointers and containers.

The yBilinear namespace relies exclusively on standard C++ operations and functions. However, example code that is included in this report makes use of the yRandom namespace[1] for generating pseudorandom numbers and the yBmp namespace[2] for creating images.

2. Derivations

2.1 Evenly Spaced Rectangular Surface Grids Defined

Surface information can be stored in evenly spaced rectangular grids, as shown in Fig. 1, where each dot represents a grid point with associated *x*, *y*, and *z* coordinates.

Since surface grids are assumed to be evenly spaced, x_i and y_j values can be stored implicitly:

$$x_i = x_0 + i\Delta x \text{ for } 0 \le i < m \tag{1}$$

and

$$y_j = y_0 + j\Delta y \text{ for } 0 \le j < n, \tag{2}$$

where x_0 and y_0 are used to locate the lower-left corner of the grid, and Δx and Δy are defined to be the difference between consecutive x and y values, respectively.

$$\Delta x \equiv x_{i+1} - x_i . \tag{3}$$

$$\Delta y \equiv y_{j+1} - y_j . \tag{4}$$

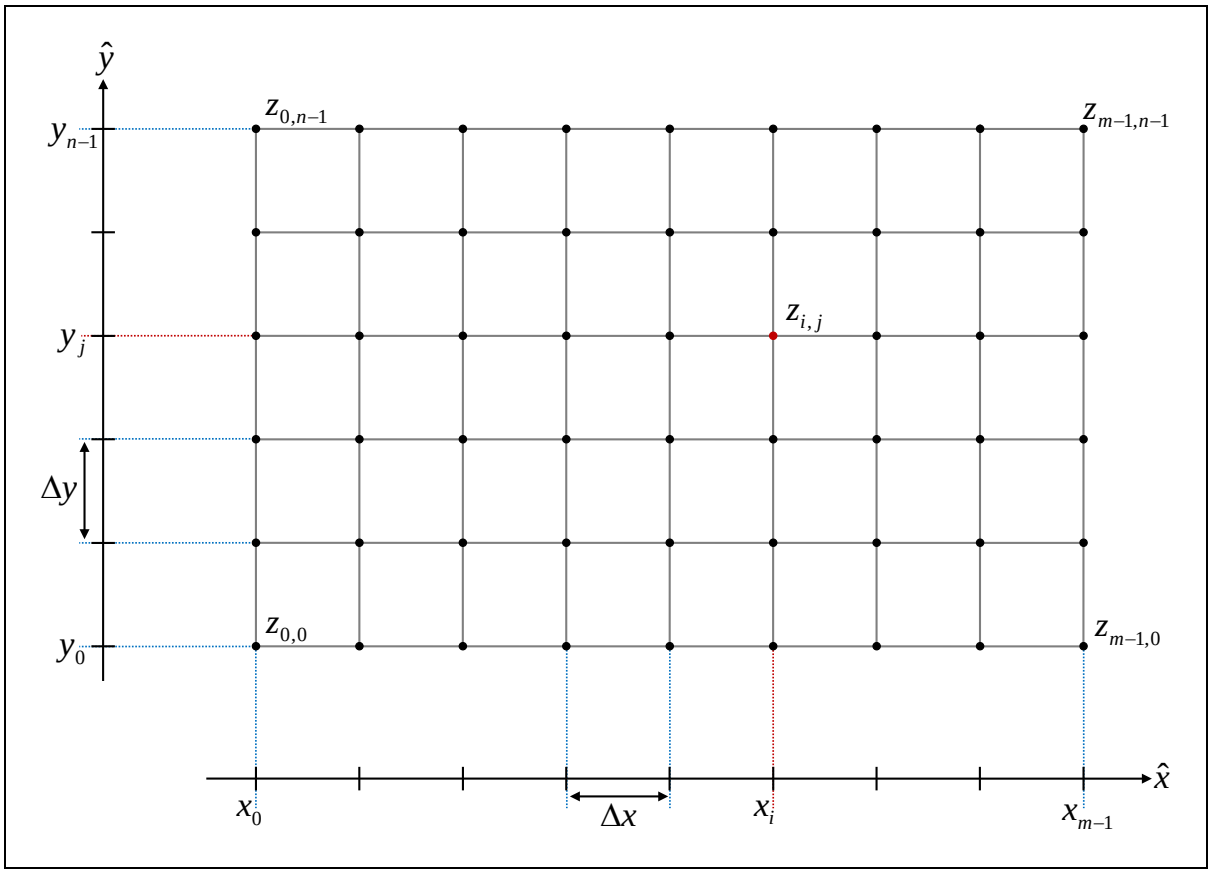


Fig. 1 An evenly spaced rectangular surface grid

Eqs. 5 and 6 can be used to find grid indices given spatial coordinates.

$$i = \text{trunc}\left(\frac{x - x_0}{\Delta x}\right) \tag{5}$$

and

$$j = \text{trunc}\left(\frac{y - y_0}{\Delta y}\right), \tag{6}$$

where the trunc() function rounds toward zero to the nearest integer.

Eqs. 7 and 8 can be used to find maximum values for the x and y coordinates.

$$x_{\max} \equiv x_{m-1} = x_0 + (m-1)\Delta x\,. \tag{7}$$

$$y_{\max} \equiv y_{n-1} = y_0 + (n-1)\Delta y\,. \tag{8}$$

2.2 Bilinear Interpolations

Bilinear interpolations can be used to estimate surface values for locations that are between grid points.

To calculate a bilinear interpolation, begin with the equation for calculating a linear interpolation, which is easily derived from the point-slope equation of a line (Fig. 2):

$$y = \frac{x - x_\alpha}{x_\beta - x_\alpha}\left(y_\beta - y_\alpha\right) + y_\alpha \,. \tag{9}$$

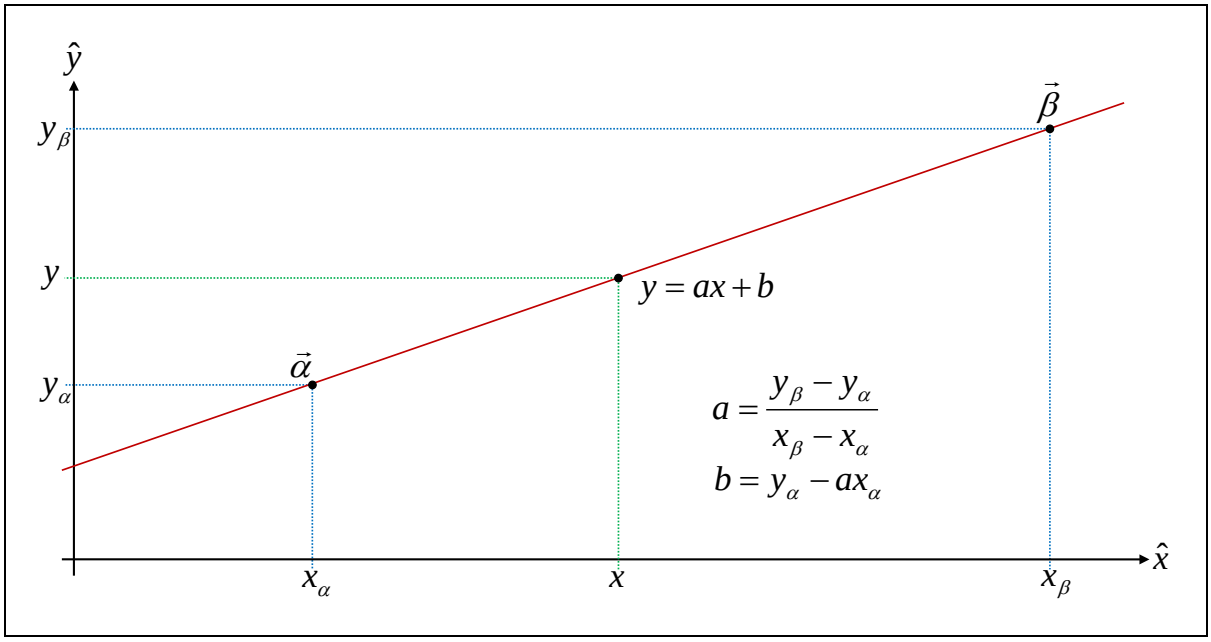


Fig. 2 A linear interpolation

A bilinear interpolation can be calculated by performing 3 linear interpolations. First, use Eq. 9 to interpolate in the $\hat{y}$ direction (shown in green in Fig. 3):

$$z_\alpha = \frac{y - y_j}{y_{j+1} - y_j}\left(z_{i,j+1} - z_{i,j}\right) + z_{i,j} \,. \tag{10}$$

$$z_\beta = \frac{y - y_j}{y_{j+1} - y_j}\left(z_{i+1,j+1} - z_{i+1,j}\right) + z_{i+1,j} \,. \tag{11}$$

Next, interpolate in the $\hat{x}$ direction (shown in red in Fig. 3):

$$z = \frac{x - x_i}{x_{i+1} - x_i}\left(z_\beta - z_\alpha\right) + z_\alpha \,. \tag{12}$$

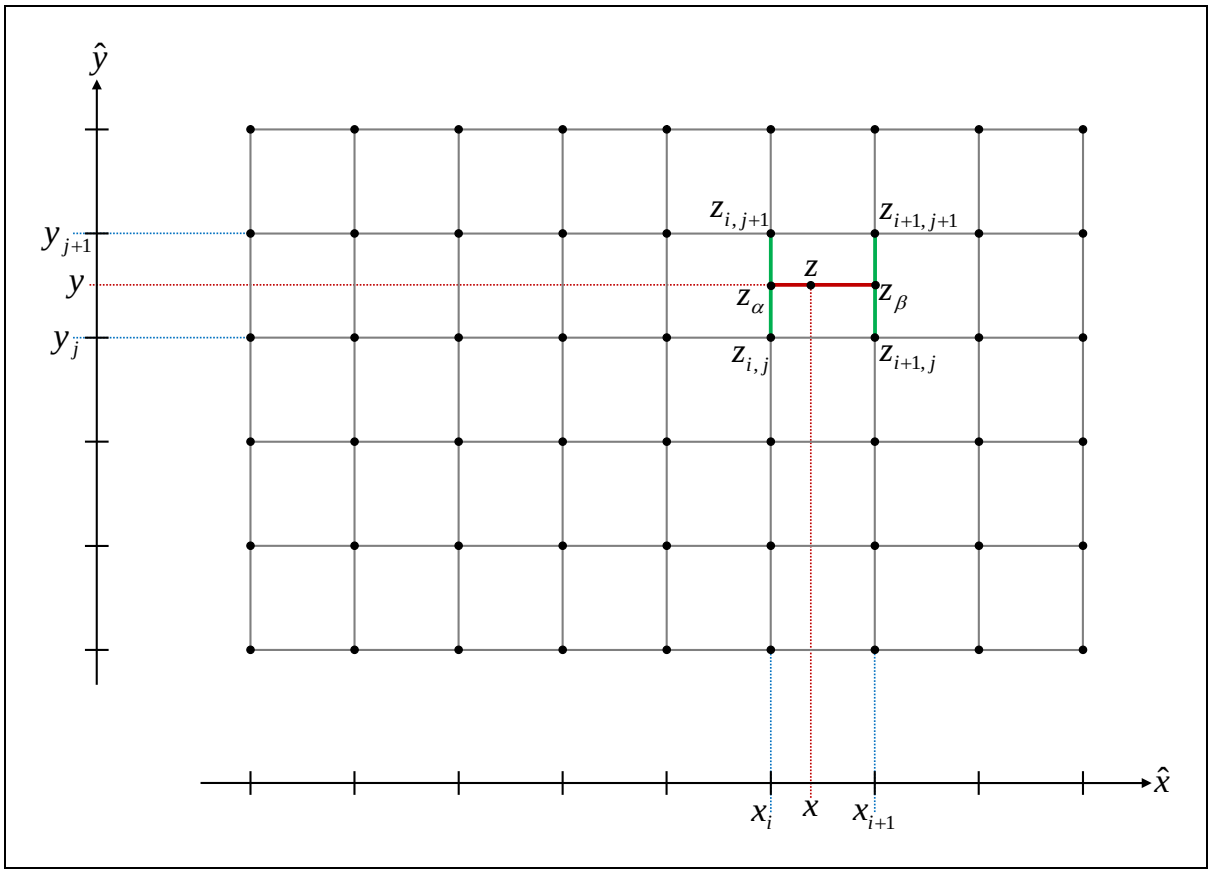


Fig. 3 A bilinear interpolation

Eqs. 2 and 4 can be used to rewrite Eqs. 10 and 11 in terms of y_0 and Δy:

$$z_\alpha = \left(\frac{y - y_0}{\Delta y} - j \right) \left(z_{i,j+1} - z_{i,j} \right) + z_{i,j} . \tag{13}$$

$$z_\beta = \left(\frac{y - y_0}{\Delta y} - j \right) \left(z_{i+1,j+1} - z_{i+1,j} \right) + z_{i+1,j} . \tag{14}$$

Similarly, Eqs. 1 and 3 can be used to rewrite Eq. 12 in terms of x_0 and Δx:

$$z = \left(\frac{x - x_0}{\Delta x} - i \right) \left(z_\beta - z_\alpha \right) + z_\alpha . \tag{15}$$

2.3 Line-Surface Intersections (Cells)

Suppose that a line, L, passes through the points $\vec{L}_0$ and $\vec{L}_1$, where

$$\vec{L}_0 = L_{0,x}\hat{x} + L_{0,y}\hat{y} + L_{0,z}\hat{z} \text{ and } \vec{L}_1 = L_{1,x}\hat{x} + L_{1,y}\hat{y} + L_{1,z}\hat{z} . \tag{16}$$

Let $\vec{p}$ represent a point that lies on L, where

$$\vec{p} = x\hat{x} + y\hat{y} + z\hat{z}\,. \tag{17}$$

$\vec{L}_0$ and $\vec{L}_1$ can be used to construct a parametric equation for $\vec{p}$ as a function of t:

$$\vec{p} = \left(\vec{L}_1 - \vec{L}_0\right)t + \vec{L}_0\,. \tag{18}$$

The parameter t represents the scaled distance from $\vec{L}_0$ to $\vec{L}_1$ along L. Thus, if $t = 0$, $\vec{p}$ is located at $\vec{L}_0$. If $t = 1$, $\vec{p}$ is located at $\vec{L}_1$.

From Eqs. 17 and 18,

$$x = C_0 t + L_{0,x} \quad , \quad y = C_1 t + L_{0,y} \quad , \text{ and } \quad z = C_2 t + L_{0,z}\,, \tag{19}$$

where

$$C_0 \equiv L_{1,x} - L_{0,x} \quad , \quad C_1 \equiv L_{1,y} - L_{0,y} \quad , \text{and } C_2 \equiv L_{1,z} - L_{0,z}\,. \tag{20}$$

Substituting Eqs. 10 and 11 into Eq. 12 and making use of Eqs. 3 and 4,

$$\begin{aligned} z(x,y) = [&(x - x_{i+1})(y - y_{j+1})z_{i,j} - (x - x_i)(y - y_{j+1})z_{i+1,j} \\ &- (x - x_{i+1})(y - y_j)z_{i,j+1} + (x - x_i)(y - y_j)z_{i+1,j+1}]\,/\,\Delta x \Delta y\,. \end{aligned} \tag{21}$$

Substituting Eq. 19 into Eq. 21 results in an equation that can be used to find t at the point where L intersects the surface:

$$\begin{aligned} C_2 t + L_{0,z} = [&(C_0 t + L_{0,x} - x_{i+1})(C_1 t + L_{0,y} - y_{j+1})z_{i,j} \\ &- (C_0 t + L_{0,x} - x_i)(C_1 t + L_{0,y} - y_{j+1})z_{i+1,j} \\ &- (C_0 t + L_{0,x} - x_{i+1})(C_1 t + L_{0,y} - y_j)z_{i,j+1} \\ &+ (C_0 t + L_{0,x} - x_i)(C_1 t + L_{0,y} - y_j)z_{i+1,j+1}]\,/\,\Delta x \Delta y\,. \end{aligned} \tag{22}$$

Rearranging terms,

$$\begin{aligned} (C_2 t + L_{0,z})\Delta x \Delta y = &(C_0 t - (x_{i+1} - L_{0,x}))(C_1 t - (y_{j+1} - L_{0,y}))z_{i,j} \\ &- (C_0 t - (x_i - L_{0,x}))(C_1 t - (y_{j+1} - L_{0,y}))z_{i+1,j} \\ &- (C_0 t - (x_{i+1} - L_{0,x}))(C_1 t - (y_j - L_{0,y}))z_{i,j+1} \\ &+ (C_0 t - (x_i - L_{0,x}))(C_1 t - (y_j - L_{0,y}))z_{i+1,j+1}\,. \end{aligned} \tag{23}$$

Another set of constants can be used to make Eq. 23 a bit more manageable:

$$C_3 \equiv x_{i+1} - L_{0,x}\,,\, C_4 \equiv y_{j+1} - L_{0,y}\,,\, C_5 \equiv x_i - L_{0,x}\,,\, C_6 \equiv y_j - L_{0,y}\,. \tag{24}$$

Thus,

$$
\begin{aligned}
(C_2 t + L_{0,z})\Delta x\Delta y = {} & (C_0 t - C_3)(C_1 t - C_4) z_{i,j} \\
& - (C_0 t - C_5)(C_1 t - C_4) z_{i+1,j} \\
& - (C_0 t - C_3)(C_1 t - C_6) z_{i,j+1} \\
& + (C_0 t - C_5)(C_1 t - C_6) z_{i+1,j+1} .
\end{aligned} \tag{25}
$$

Expanding quadratics,

$$
\begin{aligned}
(C_2 t + L_{0,z})\Delta x\Delta y = {} & (C_0 C_1 t^2 - (C_1 C_3 + C_0 C_4) t + C_3 C_4) z_{i,j} \\
& - (C_0 C_1 t^2 - (C_1 C_5 + C_0 C_4) t + C_5 C_4) z_{i+1,j} \\
& - (C_0 C_1 t^2 - (C_1 C_3 + C_0 C_6) t + C_3 C_6) z_{i,j+1} \\
& + (C_0 C_1 t^2 - (C_1 C_5 + C_0 C_6) t + C_5 C_6) z_{i+1,j+1} .
\end{aligned} \tag{26}
$$

Regrouping terms,

$$
\begin{aligned}
& C_0 C_1 (z_{i,j} - z_{i+1,j} - z_{i,j+1} + z_{i+1,j+1}) t^2 \\
& + [-(C_1 C_3 + C_0 C_4) z_{i,j} + (C_1 C_5 + C_0 C_4) z_{i+1,j} \\
& \quad + (C_1 C_3 + C_0 C_6) z_{i,j+1} - (C_1 C_5 + C_0 C_6) z_{i+1,j+1} - C_2 \Delta x\Delta y] t \\
& + C_3 C_4 z_{i,j} - C_5 C_4 z_{i+1,j} - C_3 C_6 z_{i,j+1} + C_5 C_6 z_{i+1,j+1} - L_{0,z}\Delta x\Delta y = 0 .
\end{aligned} \tag{27}
$$

Note that Eq. 27 is a quadratic in standard form:

$$
a t^2 + b t + c = 0 , \tag{28}
$$

where

$$
a = C_0 C_1 \left(z_{i,j} - z_{i+1,j} - z_{i,j+1} + z_{i+1,j+1} \right), \tag{29}
$$

$$
\begin{aligned}
b = {} & -(C_1 C_3 + C_0 C_4) z_{i,j} + (C_1 C_5 + C_0 C_4) z_{i+1,j} \\
& + (C_1 C_3 + C_0 C_6) z_{i,j+1} - (C_1 C_5 + C_0 C_6) z_{i+1,j+1} - C_2 \Delta x\Delta y ,
\end{aligned} \tag{30}
$$

and

$$
c = C_3 C_4 z_{i,j} - C_5 C_4 z_{i+1,j} - C_3 C_6 z_{i,j+1} + C_5 C_6 z_{i+1,j+1} - L_{0,z}\Delta x\Delta y . \tag{31}
$$

Finally, solving for t,

$$
t = \begin{cases}
s \mid s \in \Re & \text{for } a = 0, b = 0, \text{ and } c = 0 \\
\text{undefined} & \text{for } a = 0, b = 0, \text{ and } c \neq 0 \\
-\dfrac{c}{b} & \text{for } a = 0 \text{ and } b \neq 0 \\
\dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a} & \text{for } a \neq 0
\end{cases} . \tag{32}
$$

For the $a \neq 0$ case, if the discriminant is less than zero, then no real solutions exist, implying that L does not intersect the surface.

2.4 Line-Surface Intersections (Grids)

Testing for intersection between a line segment, L, and an evenly spaced rectangular surface grid involves checking for intersection between one or more simple cell surfaces, each defined by Eq. 21. Figure 4 shows the projection of L onto the $x-y$ plane, with the cells that L might intersect highlighted in yellow.

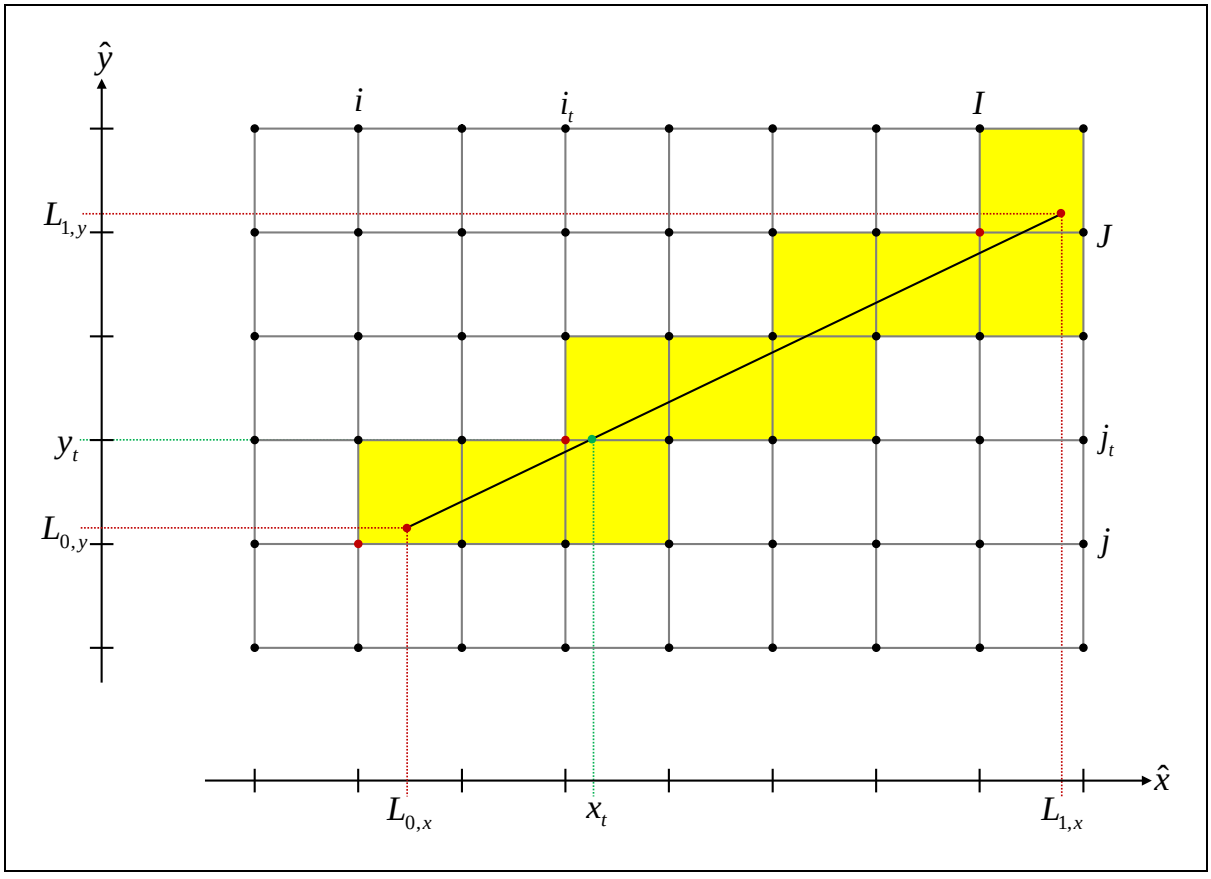


Fig. 4 Possible cell intersections for line segment L intersecting a surface

The following algorithm can be used to find indices that are associated with cells that line segment L might intersect.

1. Initial Calculations:
   a. Define i and j to be the x and y indices, respectively, of the current cell that is being checked. Use Eqs. 5 and 6, along with values for $L_{0,x}$ and $L_{0,y}$, to set initial values for i and j. Restrict initial values of $i \,|\, 0 \leq i < m-1$ and $j \,|\, 0 \leq j < n-1$.

b. Define I and J to be the x and y indices, respectively, that are associated with the end of the line segment. Use Eqs. 5 and 6, along with values for $L_{1,x}$ and $L_{1,y}$, to set I and J. Restrict values of $I \mid 0 \le I < m-1$ and $J \mid 0 \le J < n-1$.

c. Define Δi and Δj to be directional quantities that can be used to increment i and j. Use Eqs. 33 and 34 to set Δi and Δj:

$$\Delta i \equiv \begin{cases} 1 & \text{for } i < I \\ -1 & \text{otherwise} \end{cases} . \tag{33}$$

$$\Delta j \equiv \begin{cases} 1 & \text{for j} < J \\ -1 & \text{otherwise} \end{cases} . \tag{34}$$

2. Outer Loop:

a. Define i_t to be the index associated with the location where the projection of L onto the $x-y$ plane crosses a constant-*y* reference line (e.g., the location of the green dot in Fig. 4).

b. If $j = J$, set $i_t = I$.

Otherwise, use Eqs. 35 and 36 to calculate y_t then x_t. Use x_t, along with Eq. 5, to calculate i_t. Restrict the value of $i_t \mid 0 \le i_t < m-1$.

$$y_t = \begin{cases} (j+1)\Delta y + y_0 & \text{for } \Delta j = 1 \\ j\Delta y + y_0 & \text{otherwise} \end{cases} . \tag{35}$$

$$x_t = \frac{y_t - L_{0,y}}{L_{1,y} - L_{0,y}} \left(L_{1,x} - L_{0,x} \right) + L_{0,x} \quad . \tag{36}$$

3. Inner Loop:

a. Store i and j.

b. If $\Delta i (i - i_t) \le 0$, increment i by Δi then repeat step 3a. Otherwise, continue to step 4.

4. End Condition:

a. Increment j by Δj and i by $-\Delta i$.

b. If $\Delta j (J - j) \ge 0$, then jump back to step 2b. Otherwise, the algorithm is complete. The stored values for i and j represent the complete set of cells that line segment L may intersect.

2.5 Surface Gradients and Normal Vectors

To avoid confusion between the dependent variable $z(x,y)$ and the independent variable z, define $\phi(x,y) = z(x,y)$. Then, from Eq. 21,

$$\begin{aligned}\varphi(x,y) = [&(x-x_{i+1})(y-y_{j+1})z_{i,j} - (x-x_i)(y-y_{j+1})z_{i+1,j} \\ &-(x-x_{i+1})(y-y_j)z_{i,j+1} + (x-x_i)(y-y_j)z_{i+1,j+1}]/\Delta x\Delta y\,.\end{aligned} \tag{37}$$

From the definition of the del operator,

$$\vec{\nabla} f(x,y,z) = \frac{\partial f}{\partial x}\hat{x} + \frac{\partial f}{\partial y}\hat{y} + \frac{\partial f}{\partial z}\hat{z} \tag{38}$$

$$\begin{aligned}\Rightarrow \vec{\nabla}\varphi = &[(y-y_{j+1})(z_{i,j} - z_{i+1,j})/\Delta x\Delta y - (y-y_j)(z_{i,j+1} - z_{i+1,j+1})/\Delta x\Delta y]\hat{x} \\ &+[(x-x_{i+1})(z_{i,j} - z_{i,j+1})/\Delta x\Delta y - (x-x_i)(z_{i+1,j} - z_{i+1,j+1})/\Delta x\Delta y]\hat{y}\,.\end{aligned} \tag{39}$$

Thus, Eq. 39 is the gradient of the surface defined by Eq. 21. Note that $\vec{\nabla}\phi$ is not of unit length.

A number of useful surface properties can easily be calculated based on the gradient:[3]

The $x-y$ components of the direction of steepest ascent are given by $(\nabla\phi)_x$ and $(\nabla\phi)_y$ from Eq. 39.

The slope in the direction of steepest ascent is given by

$$\left|\vec{\nabla}\phi\right| = \sqrt{(\nabla\phi)_x{}^2 + (\nabla\phi)_y{}^2}\,, \tag{40}$$

and the unit-normal vector $\hat{n}$ is given by

$$\hat{n} = \frac{-(\nabla\phi)_x\hat{x} - (\nabla\phi)_y\hat{y} + \hat{z}}{\sqrt{(\nabla\phi)_x{}^2 + (\nabla\phi)_y{}^2 + 1}}\,. \tag{41}$$

Recall that unit-normal surface vectors are not unique. Specifically, if $\hat{n}$ is a unit-normal surface vector, then $-\hat{n}$ is also a unit-normal surface vector. $\hat{n}$ has been chosen to have a positive $\hat{z}$ component.

2.6 Scaled Coordinates

Eqs. 42 and 43 provide a means to convert to and from scaled coordinates.

$$s_x \equiv \frac{x - x_0}{\Delta x} \text{ and } s_y \equiv \frac{y - y_0}{\Delta y}\,. \tag{42}$$

$$x = x_0 + s_x\Delta x \text{ and } y = y_0 + s_y\Delta y\,. \tag{43}$$

The benefit of working with scaled coordinates is that they simplify calculations by introducing unit intervals and zero offsets. Thus, they allow for making the following substitutions when working with equations prior to Eq. 42:

$$x \to s_x,\ y \to s_y,\ \Delta x \to 1,\ \Delta y \to 1,\ x_0 \to 0,\ y_0 \to 0,\ x_i \to i, \text{ and } y_j \to j. \tag{44}$$

The use of scaled coordinates in the code that follows has 2 advantages: they make the code run slightly faster, and they reduce the required number of user-supplied parameters.

---

3. Calculating Indices: The SafeIndex() Function

The SafeIndex() function uses Eq. 5 (or 6), as well as the substitutions from Eq. 44, to calculate the grid index that is associated with a user-supplied scaled coordinate. The grid index is then modified to be compatible with other functions described in this report:

$$k = \begin{cases} 0 & \text{for } s < 0 \\ m-2 & \text{for } s \geq m-2. \\ \text{trunc}(s) & \text{otherwise} \end{cases} \tag{45}$$

3.1 SafeIndex() Code

```
inline int SafeIndex(//<=====CALCULATE A SAFE INDEX AT A GIVEN SCALED LOCATION
    int m,//<------------------NUMBER OF GRID INDICES (m SHOULD BE AT LEAST 2)
    double s){//<-------------------------A GRID LOCATION IN SCALED COORDINATES
  return s<0?0:s>m-2?m-2:int(s);
}//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

3.2 SafeIndex() Parameters

m — **m** specifies the array size that is associated with the scaled distance **s**. Typically, **m** refers to m when **s** = s_x and n when **s** = s_y. **m** should be greater than 1.

s — **s** specifies a scaled distance, typically either s_x or s_y from Eq. 42.

3.3 SafeIndex() Return Value

The SafeIndex() function returns the index k from Eq. 45.

4. Performing Interpolations: The Interpolate() Function

The Interpolate() function uses Eqs. 13, 14, and 15, as well as the substitutions from Eq. 44, to perform bilinear interpolations. The equations have been coded in a slightly more compact form:

$$z_\alpha = (s_y - j)(z_{i,j+1} - z_{i,j}) + z_{i,j}. \tag{46}$$

$$z = (s_x - i)\{(s_y - j)(z_{i+1,j+1} - z_{i+1,j}) + z_{i+1,j} - z_\alpha\} + z_\alpha. \tag{47}$$

4.1 Interpolate() Code

```
template<class T>double Interpolate(//<=================BILINEAR INTERPOLATION
    const T&Z,//<----------------------POINTER TO Z WHERE Z[i][j]=Z(X[i],Y[j])
    int i,int j,//<-------------------------GRID INDICES (0<=i<m-1 & 0<=j<n-1)
    double sx,double sy){//<-------------A GRID LOCATION IN SCALED COORDINATES
  double za=(sy-j)*(Z[i][j+1]-Z[i][j])+Z[i][j];
  return (sx-i)*((sy-j)*(Z[i+1][j+1]-Z[i+1][j])+Z[i+1][j]-za)+za;
}//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

4.2 Interpolate() Parameters

Z — **Z** points to a 2-index array of values that represents $z(x, y)$, where **Z**[**i**][**j**] = $z(x_i, y_j)$.

i — **i** specifies i, an index that is associated with the x axis. Values for **i** cannot be less than zero or greater than $m-2$, where m is the first-index size of **Z**.

j — **j** specifies j, an index that is associated with the y axis. Values for **j** cannot be less than zero or greater than $n-2$, where n is the second-index size of **Z**.

sx — **sx** specifies s_x, a scaled distance from Eq. 42.

sy — **sy** specifies s_y, a scaled distance from Eq. 42.

4.3 Interpolate() Return Value

The Interpolate() function returns z from Eq. 47.

4.4 Interpolate() Simple Example

The following example begins by defining the surface that is shown in Fig. 5. The Interpolate() function is then used to estimate z at $x = 1.5$ and $y = 0.5$.

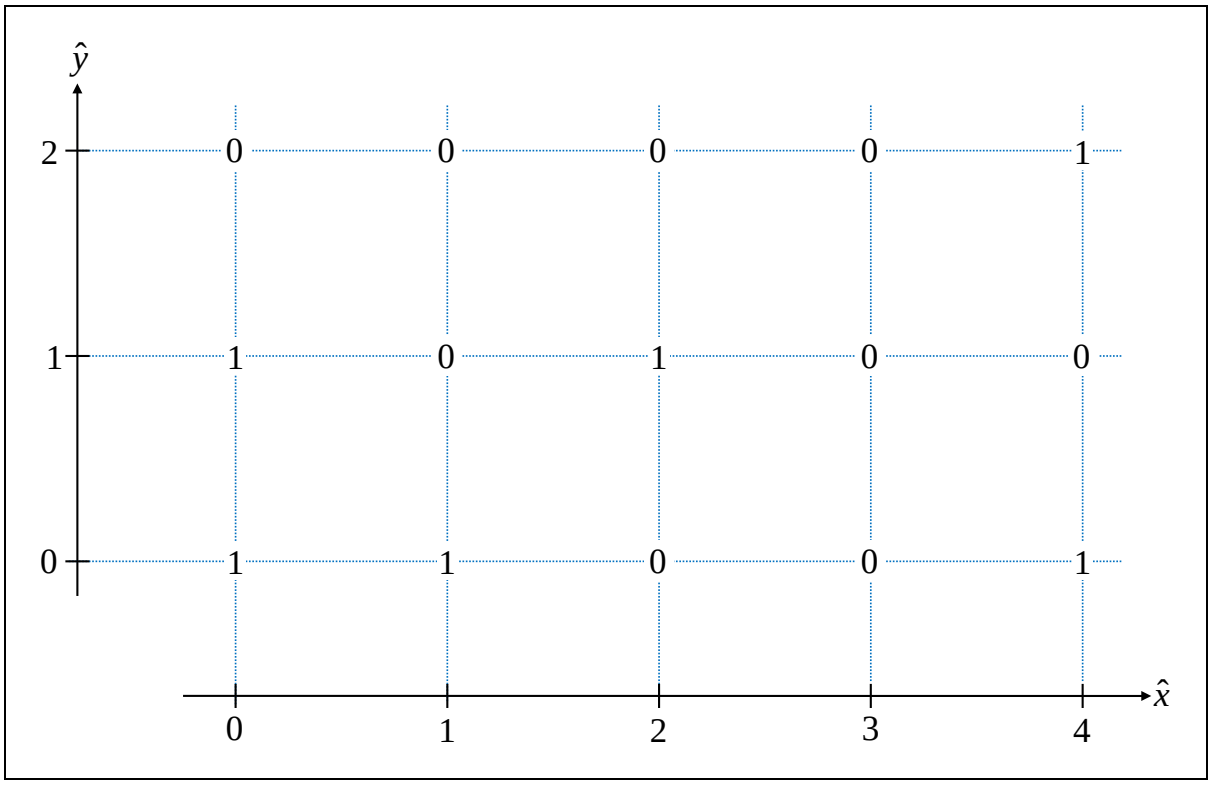


Fig. 5 Example surface

```
#include <cstdio>//.....................................................printf()
#include "y_bilinear.h"//.............................................yBilinear
int main(){//<=================A SIMPLE EXAMPLE USING THE Interpolate() FUNCTION
  const int m=5,n=3;
  double Z[m][n]={1,1,0 , 1,0,0 , 0,1,0 , 0,0,0 , 1,0,1};
  double x=1.5,y=.5;
  int i=yBilinear::SafeIndex(m,x),j=yBilinear::SafeIndex(n,y);
  double z=yBilinear::Interpolate(Z,i,j,x,y);
  printf("At x=%.1f and y=%.1f, z=%.1f\n",x,y,z);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

OUTPUT:

```
At x=1.5 and y=0.5, z=0.5
```

4.5 Interpolate() Image Example

The following example begins by defining the Rainbow() function, which can be used to map numbers to colors. Next, the example defines the surface that is shown in Fig. 5. Functions from the yBmp namespace, along with the Interpolate() function, are used to create a pseudo-color image of the surface. Finally, the image is written to a BMP file, which is displayed in Fig. 6. The color-to-value key is shown in Fig. 7.

```
inline void Rainbow(//<=======================================RAINBOW COLOR MAP
    unsigned char C[3],//<----------------------------OUTPUT COLOR (CALCULATED)
    double x,//<----------------------VALUE FOR WHICH A COLOR WILL BE CALCULATED
    double min,double max){//<-----------------MINIMUM AND MAXIMUM SCALED VALUES
  if(x<min){C[0]=C[1]=C[2]=0;/*&*/return;}//.......set too small values to black
  if(x>max){C[0]=C[1]=C[2]=255;/*&*/return;}//.....set too large values to white
  x=(1-(x-min)/(max-min))*8;//.....................remap x to a range of 8 to 0
  C[0]=int((3<x&&x<5||x>7     ?-fabs(x/2-3)+1.5:5<=x&&x<=7?1:0)*255);//.....blue
  C[1]=int((1<x&&x<3||5<x&&x<7?-fabs(x/2-2)+1.5:3<=x&&x<=5?1:0)*255);//....green
  C[2]=int((     x<1||3<x&&x<5?-fabs(x/2-1)+1.5:1<=x&&x<=3?1:0)*255);//......red
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~04JUN2014~~~~~~
```

```
#include "y_bmp.h"//.....................................................yBmp
#include "y_bilinear.h"//.............................yBilinear,<cmath>{fabs()}
int main(){//<================CREATE AN IMAGE FROM A SURFACE USING Interpolate()
  const int m=5,n=3,M=1000,N=500;
  double Z[m][n]={1,1,0 , 1,0,0 , 0,1,0 , 0,0,0 , 1,0,1};
  unsigned char*I=yBmp::NewImage(M,N,255);
  for(int q=0;q<M;++q)for(int p=0;p<N;++p){
    double x=q*(m-1)*1./(M-1),y=p*(n-1)*1./(N-1);
    int i=yBilinear::SafeIndex(m,x),j=yBilinear::SafeIndex(n,y);
    double z=yBilinear::Interpolate(Z,i,j,x,y);
    Rainbow(yBmp::GetPixel(I,q,p),z,0,1);}
  yBmp::WriteBmpFile("interpolate.bmp",I);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

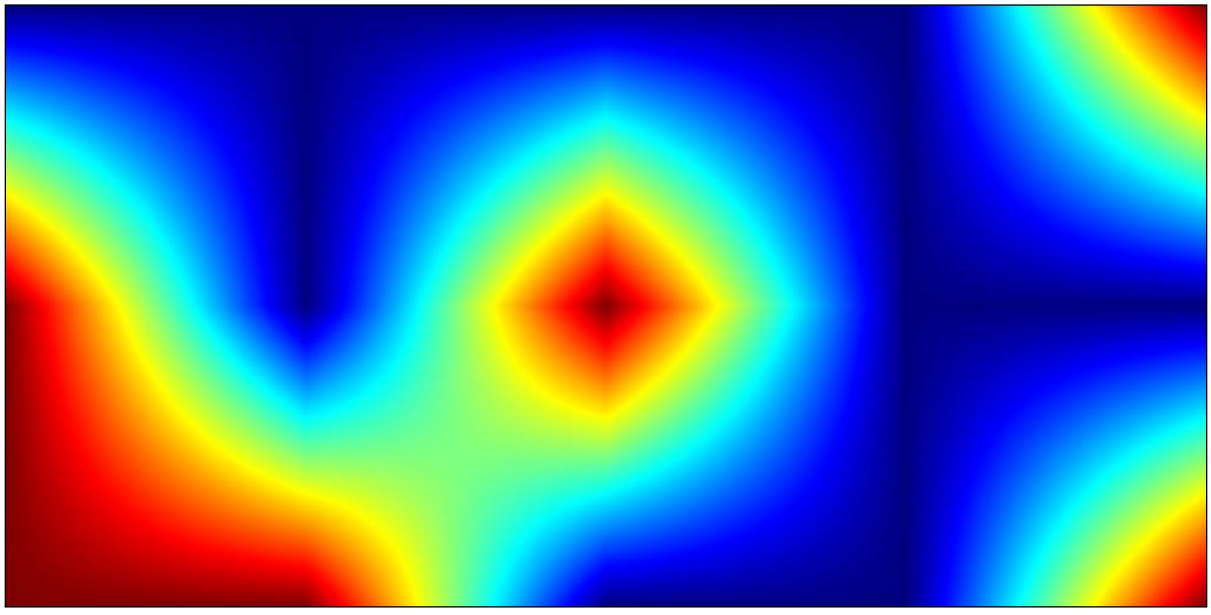


Fig. 6 Image generated by example code from Section 4.5.

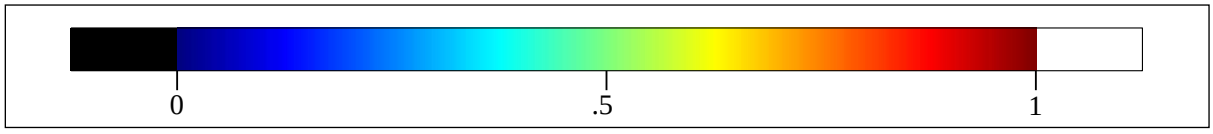


Fig. 7 Color-to-value key

5. Calculating Line-Cell Intersections: The CellIntersect() Function

The CellIntersect() function uses equations from Section 2.3 to calculate the point where a line intersects a simple surface that is defined by Eq. 21. To simplify the CellIntersect() function, the equations from Section 2.3 have to been rewritten in terms of scaled coordinates:

Eqs. 20 and 24 can be converted to scaled coordinates using the substitutions given in Eq. 44:

$$C_0 = L_{1,s_x} - L_{0,s_x} \quad , \quad C_1 = L_{1,s_y} - L_{0,s_y} \quad , \quad C_2 = L_{1,z} - L_{0,z} \, . \tag{48}$$

$$C_3 = i + 1 - L_{0,s_x} \quad , \quad C_4 = j + 1 - L_{0,s_y} \quad , \quad C_5 = i - L_{0,s_x} \quad , \quad C_6 = j - L_{0,s_y} \, . \tag{49}$$

The same is true for Eqs. 29, 30, and 31:

$$a = C_0 C_1 \left(z_{i,j} - z_{i+1,j} - z_{i,j+1} + z_{i+1,j+1} \right), \tag{50}$$

$$\begin{aligned} b = & -(C_1 C_3 + C_0 C_4) z_{i,j} + (C_1 C_5 + C_0 C_4) z_{i+1,j} \\ & + (C_1 C_3 + C_0 C_6) z_{i,j+1} - (C_1 C_5 + C_0 C_6) z_{i+1,j+1} - C_2 \, , \end{aligned} \tag{51}$$

and

$$c = C_3 C_4 z_{i,j} - C_5 C_4 z_{i+1,j} - C_3 C_6 z_{i,j+1} + C_5 C_6 z_{i+1,j+1} - L_{0,z} \, . \tag{52}$$

Scaled coordinates at the point of intersection can be found by substituting t from Eq. 32 into Eq. 19 (after converting to scaled coordinates):

$$s_x = C_0 t + L_{0,s_x} \quad , \quad s_y = C_1 t + L_{0,s_y} \quad , \quad z = C_2 t + L_{0,z} \, . \tag{53}$$

Note that line L will intersect the surface at 0, 1, 2, or infinitely many locations.

If L intersects the surface at 2 locations, then the rules for selecting which intersection to choose are somewhat complicated: The CellIntersect() function attempts to find the first intersection, when traveling from $\vec{L}_0$ toward $\vec{L}_1$, that is within the bounds given by $x_i \le s_x \le x_{i+1}$ and $y_j \le s_y \le y_{j+1}$. If no intersection is found, then the function attempts to find the first intersection when traveling from $\vec{L}_0$ toward $\vec{L}_1$ (regardless of whether or not it is within the cell's boundaries). If an intersection still isn't found, then the function uses the first intersection when traveling from $\vec{L}_0$ away from $\vec{L}_1$.

5.1 CellIntersect() Code

```
template<class T>int CellIntersect(//<==================LINE-CELL INTERSECTION
    const T&Z,//<----------------------POINTER TO Z WHERE Z[i][j]=Z(X[i],Y[j])
    int i,int j,//<--------------------------GRID INDICES (0<=i<m-1 & 0<=j<n-1)
    const double L[6],//<----------A SCALED LINE {L0SX,L0SY,L0Z,L1SX,L1SY,L1Z}
    double&sx,double&sy,double&z,//<----SCALED INTERSECTION POINT (CALCULATED)
    double epsilon=1E-9){//<------------------VALUE FOR DIVISION-BY-ZERO CHECK
  double C0=L[3]-L[0],C1=L[4]-L[1],C2=L[5]-L[2];
  double C5=i-L[0],C6=j-L[1],C3=C5+1,C4=C6+1;
  double a=C0*C1*(Z[i][j]-Z[i+1][j]-Z[i][j+1]+Z[i+1][j+1]),
    b=-(C1*C3+C0*C4)*Z[i][j]+(C1*C5+C0*C4)*Z[i+1][j]
      +(C1*C3+C0*C6)*Z[i][j+1]-(C1*C5+C0*C6)*Z[i+1][j+1]-C2,
    c=C3*C4*Z[i][j]-C5*C4*Z[i+1][j]-C3*C6*Z[i][j+1]+C5*C6*Z[i+1][j+1]-L[2];
  double t,D;
  if(fabs(a)<epsilon&&fabs(b)<epsilon)return fabs(c)<epsilon?-1:0;
  if(fabs(a)<epsilon)t=-c/b,sx=C0*t+L[0],sy=C1*t+L[1],z=C2*t+L[2];
  else{
    if((D=b*b-4*a*c)<0)return 0;//..no intersection, <sx,sy,sz> not calculated
    t=(-b-sqrt(D))/2/a,sx=C0*t+L[0],sy=C1*t+L[1],z=C2*t+L[2];
    double tb=(-b+sqrt(D))/2/a,sxb=C0*tb+L[0],syb=C1*tb+L[1],zb=C2*tb+L[2];
    if(t>=0&&t<=1&&tb>=0&&tb<=1){
      bool ra=sx<i||sx>i+1||sy<j||sy>j+1?1:0;
      bool rb=sxb<i||sxb>i+1||syb<j||syb>j+1?1:0;
      if(ra&&rb&&t>tb){sx=sxb,sy=syb,z=zb;return 1;}
      else if(ra||!rb&&t>tb)t=tb,sx=sxb,sy=syb,z=zb;}
    else if(t>tb&&tb>0||t<=0&&t<tb)t=tb,sx=sxb,sy=syb,z=zb;}
  if(sx<i||sx>i+1||sy<j||sy>j+1)return 1;//.....................out of bounds
  return t<0?2:(t>1?3:4);//....line (2), ray (3), or segment (4) intersections
}//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

5.2 CellIntersect() Parameters

Z — **Z** points to a 2-index array of values that represents $z(x, y)$, where **Z**[**i**][**j**] = $z(x_i, y_j)$.

i — **i** specifies i, an index that is associated with the x axis. Values for **i** cannot be less than zero or greater than $m-2$, where m is the first-index size of **Z**.

j — **j** specifies j, an index that is associated with the y axis. Values for **j** cannot be less than zero or greater than $n-2$, where n is the second-index size of **Z**.

L — **L** specifies 2 points that define a line (**L**={ $\vec{L}_{0,s_x}, \vec{L}_{0,s_y}, \vec{L}_{0,z}, \vec{L}_{1,s_x}, \vec{L}_{1,s_y}, \vec{L}_{1,z}$ }).

sx — **sx** is the calculated value for s_x, the point of intersection between **L** and the surface in scaled coordinates (see Eq. 53). If the function's return value is –1 or 0, then **sx** is not calculated.

sy — **sy** is the calculated value for s_y, the point of intersection between **L** and the surface in scaled coordinates (see Eq. 53). If the function's return value is –1 or 0, then **sy** is not calculated.

z — **z** is the calculated value for z, the point of intersection between **L** and the surface (see Eq. 53). If the function's return value is –1 or 0, then **z** is not calculated.

epsilon — **epsilon** specifies the minimum nonzero value for both a from Eq. 50 and b from Eq. 51. Thus, if $|a|$ is less than **epsilon**, then a is considered to be zero, and if $|b|$ is less than **epsilon**, then b is considered to be zero.

5.3 CellIntersect() Return Value

In addition to (possibly) calculating values for **sx**, **sy**, and **z**, the CellIntersect() function returns 1 of 6 values (–1, 0, 1, 2, 3, or 4):

- A return value of –1 indicates that a, b, and c from Eq. 28 are all effectively zero. This means that line L intersects the surface at an infinite number of points. For a return value of –1, sx, sy, and z are not calculated.
- A return value of 0 indicates that the line defined by **L** does not intersect the surface at any point (and, thus, **sx**, **sy**, and **z** are not calculated).
- A return value of 1 indicates that the line defined by **L** intersects the surface at a point that is outside the boundaries of the cell.
- A return value of 2 (or greater) indicates that the line defined by **L** intersects the surface within the bounds of the cell.
- A return value of 3 (or greater) indicates that the ray defined by **L** (from $\vec{L}_0$ to $\vec{L}_1$) intersects the surface within the bounds of the cell.
- A return value of 4 indicates that the line segment defined by **L** intersects the surface within the bounds of the cell.

5.4 CellIntersect() Simple Example

The following example begins by defining the surface that is shown in Fig. 5. The CellIntersect() function is then used to determine the point of intersection between the surface and a vertical line located at $x = 1.5$, $y = 0.5$.

```
#include <cstdio>//.....................................................printf()
#include "y_bilinear.h"//.............................................yBilinear
int main(){//<===============A SIMPLE EXAMPLE USING THE CellIntersect() FUNCTION
  const int m=5,n=3;
  double Z[m][n]={1,1,0 , 1,0,0 , 0,1,0 , 0,0,0 , 1,0,1};
  int i=yBilinear::SafeIndex(m,1.5),j=yBilinear::SafeIndex(n,.5);
  double L[6]={1.5,.5,1,1.5,.5,0};
  double x,y,z;
  int intersect=yBilinear::CellIntersect(Z,i,j,L,x,y,z);
  if(intersect==-1)printf("Intersection cannot be determined.\n");
  else if(intersect==0)printf("The line does not intersect the surface.\n");
  else printf("Intersection at x=%.1f, y=%.1f, and z=%.1f.\n",x,y,z);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

OUTPUT:

```
Intersection at x=1.5, y=0.5, and z=0.5.
```

5.5 CellIntersect() Image Example

The following example begins by defining the surface that is shown in Fig. 5. Note that the surface is a composite of 8 simple surfaces that are each described by Eq. 21. Next, the CellIntersect() function is used to calculate the point of intersection between the simple surface that corresponds with the $i = 0$, $j = 0$ cell and 10^7 randomly chosen lines. Functions from the yBmp namespace, along with the Rainbow() function (presented in Section 4.5), are used to create a pseudo-color image of the intersections. Finally, the image is written to a BMP file, which is displayed in Fig. 8. The color-to-value key is shown in Fig. 7. Since the intersecting lines are randomly generated, not all points are represented. Non-plotted points are shown in white.

```
#include <cstdlib>//.............................................rand(),RAND_MAX
#include "y_bmp.h"//.....................................................yBmp
#include "y_bilinear.h"//............................yBilinear,<cmath>{fabs()}
int main(){//<==============CREATE AN IMAGE FROM A SURFACE USING CellIntersect()
  const int m=5,n=3,M=1000,N=500;
  double Z[m][n]={1,1,0 , 1,0,0 , 0,1,0 , 0,0,0 , 1,0,1};
  unsigned char*I=yBmp::NewImage(M,N,255);
  for(int i=0;i<10000000;++i){
    double L[6]={rand()*(m-1)*1./RAND_MAX,
                 rand()*(n-1)*1./RAND_MAX,
                 -5+rand()*10./RAND_MAX,
                 rand()*(m-1)*1./RAND_MAX,
                 rand()*(n-1)*1./RAND_MAX,
                 -5+rand()*10./RAND_MAX};
    double x,y,z;
    if(yBilinear::CellIntersect(Z,0,0,L,x,y,z)>0){
      int q=int(x/(m-1)*M),p=int(y/(n-1)*N);
      if(q>=0&&q<M&&p>=0&&p<N)Rainbow(yBmp::GetPixel(I,q,p),z,0,1);}}
  yBmp::WriteBmpFile("cell_intersect.bmp",I);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

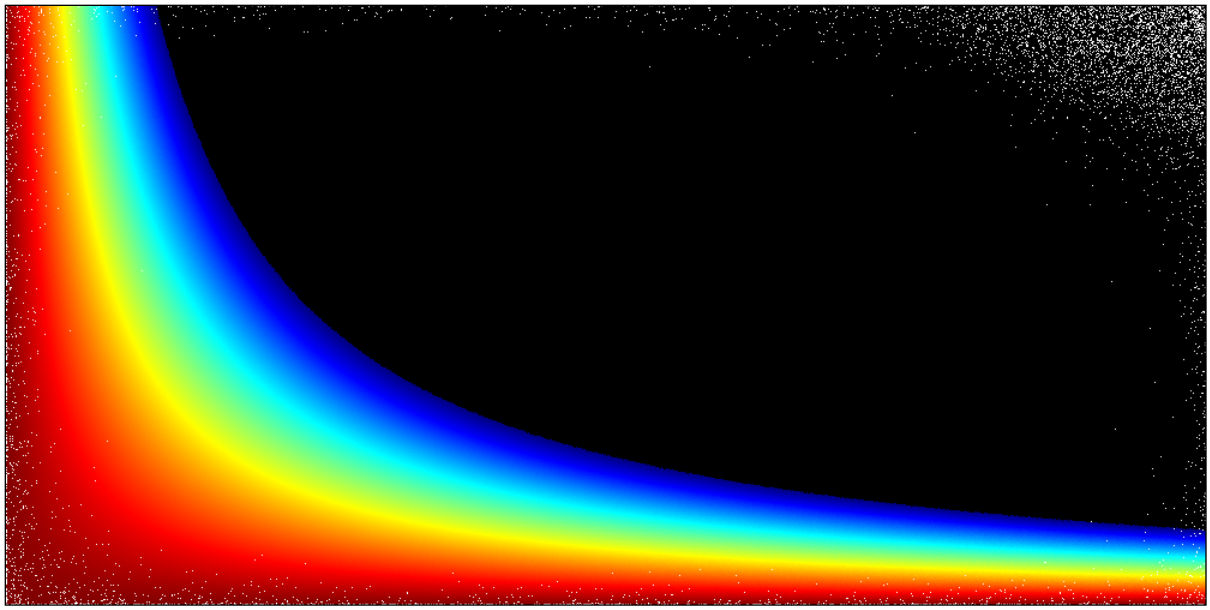


Fig. 8 Image generated by example code from Section 5.5.

6. Calculating Line-Surface Intersections: The Line() Function

The Line() function is designed to work with the CellIntersect() function to determine the point of intersection (if it exists) between a line segment, L, and an evenly spaced rectangular surface grid. This is accomplished by first using the Line() function to find a set of cells that L may intersect, then using the CellIntersect() function to test each of the cells for intersection.

The set of cells specified by the Line() function is typically much smaller than the set of all cells that makes up a surface. Thus, the primary purpose of the Line() function is to reduce the number of cells that must be tested using the CellIntersect() function. For some scenarios, it may be possible to further reduce the number of cells that need to be tested by performing additional tests after calling the Line() function but before calling the CellIntersect() function.

The Line() function uses the algorithm presented in Section 2.4 to determine the indices of the cells that line segment L might intersect when traveling from L_0 to L_1. However, the algorithm has been optimized in 2 ways.

Eq. 36, which is used to find i_t, can be written to allow for some calculations to be removed from the outer loop:

$$x_t = (j + b_j)M + P \quad , \tag{54}$$

where

$$b_j \equiv \begin{cases} 1 & \text{for } \Delta j = 1 \\ 0 & \text{otherwise} \end{cases}, \tag{55}$$

$$M \equiv \begin{cases} \dfrac{L_{1,x} - L_{0,x}}{L_{1,y} - L_{0,y}} & \text{for } j \neq J \\ 0 & \text{otherwise} \end{cases}, \tag{56}$$

and

$$P \equiv L_{0,x} - L_{1,x} M \ . \tag{57}$$

The Line() function has been written to take advantage of the fact that the algorithm presented in Section 2.4 is more efficient for lines whose slopes are closer to horizontal than vertical. If $|I - i| > |J - j|$ then the algorithm is used as described. Otherwise, the location where the projection of L onto the $x-y$ plane crosses a constant x reference line (rather than a constant y reference line) is found. This results in modified outer and inner loops that are very similar to those in Section 2.4, but where indices and measured values associated with the x and y axes have been exchanged.

6.1 Line() Code

```
 inline int Line(//<====================================LINE-GRID INTERSECTION
     int m,int n,//<-----NUMBERS OF GRID INDICES (m AND n SHOULD BE AT LEAST 2)
     const double L[6],//<--A SCALED LINE SEGMENT {L0SX,L0SY,L0Z,L1SX,L1SY,L1Z}
     int*A,int*B){//<--------STORAGE FOR CELL INDICES (FOR EACH, SIZE >= m+n-3)
   int i=SafeIndex(m,*L),j=SafeIndex(n,L[1]),I=SafeIndex(m,L[3]),
     J=SafeIndex(n,L[4]),di=i<I?1:-1,dj=j<J?1:-1,bi=di>0?1:0,bj=dj>0?1:0,k=0;
   double M=J-j?(L[3]-*L)/(L[4]-L[1]):0,N=I-i?1/M:0,P=*L-L[1]*M,Q=-P*N;
   if(abs(I-i)>abs(J-j))for(int it;dj*(J-j)>=0;j+=dj,i-=di)//dj* selects < or >
     for(it=j-J?SafeIndex(m,(j+bj)*M+P):I;di*(i-it)<=0;i+=di)A[k]=i,B[k++]=j;
   else for(int jt;di*(I-i)>=0;i+=di,j-=dj)//.................di* selects < or >
     for(jt=i-I?SafeIndex(n,(i+bi)*N+Q):J;dj*(j-jt)<=0;j+=dj)A[k]=i,B[k++]=j;
   return k;//.......................the number of elements to check in A and B
 }//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

6.2 Line() Parameters

m — **m** specifies m, the number of horizontal values that the surface grid contains. **m** must be greater than 1.

n — **n** specifies n, the number of vertical values that the surface grid contains. **n** must be greater than 1.

L — **L** specifies 2 points that define a line segment (**L**={ $L_{0,s_x}, L_{0,s_y}, L_{0,z}, L_{1,s_x}, L_{1,s_y}, L_{1,z}$ }).

A — **A** points to storage for *x*-axis indices. The size of **A** must be at least **m** + **n** – 3.

B — **B** points to storage for *y*-axis indices. The size of **B** must be at least **m** + **n** – 3.

6.3 Line() Return Value

In addition to calculating values for **A** and **B**, the Line() function returns the number of index pairs that are generated. The indices found in **A** and **B** are stored in the order in which they are found when traveling from L_0 to L_1.

6.4 Line() Simple Example

The following example begins by defining the surface that is shown in Fig. 5. The Line() and CellIntersect() functions are then used to determine the point of intersection between the surface and a vertical line located at $x = 1.5$, $y = 0.5$.

```
#include <cstdio>//.....................................................printf()
#include "y_bilinear.h"//.............................................yBilinear
int main(){//<=========================A SIMPLE EXAMPLE USING THE Line() FUNCTION
  const int m=5,n=3;
  double x0=0,y0=0,dx=1,dy=1,Z[m][n]={1,1,0 , 1,0,0 , 0,1,0 , 0,0,0 , 1,0,1};
  double L[6]={1.5,.5,1,1.5,.5,0};
  int *A=new int[m+n],*B=new int[m+n],k=yBilinear::Line(m,n,L,A,B);
  double x,y,z;
  bool b=0;
  for(int j=0,q;j<k;++j)
    if((q=yBilinear::CellIntersect(Z,A[j],B[j],L,x,y,z))>2){
      b=!(q-4);break;}
  printf("Intersection at x=%.1f, y=%.1f, and z=%.1f.\n",x,y,z);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

OUTPUT:

```
Intersection at x=1.5, y=0.5, and z=0.5.
```

6.5 Line() Image Example

The following example begins by defining the surface that is shown in Fig. 5. Next, the Line() and CellIntersect() functions are used to calculate the point of intersection between the surface and 10^7 randomly chosen line segments. Functions from the yBmp namespace, along with the Rainbow() function (presented in Section 4.5), are used to create a pseudo-color image of the intersections. Finally, the image is written to a BMP file, which is displayed in Fig. 9. The color-to-value key is shown in Fig. 7. Since intersecting lines are randomly generated, not all points are represented. Non-plotted points are shown in white.

```
#include <cstdlib>//.............................................rand(),RAND_MAX
#include "y_bmp.h"//.....................................................yBmp
#include "y_bilinear.h"//............................yBilinear,<cmath>{fabs()}
int main(){//<=======================CREATE AN IMAGE FROM A SURFACE USING Line()
  const int m=5,n=3,M=1000,N=500;
  double Z[m][n]={1,1,0 , 1,0,0 , 0,1,0 , 0,0,0 , 1,0,1};
  unsigned char*I=yBmp::NewImage(M,N,255);
  int K=0,J=0,*A=new int[m+n],*B=new int[m+n];
  for(int i=0;i<10000000;++i){
    double L[6]={rand()*(m-1)*1./RAND_MAX,
```

```
                rand()*(n-1)*1./RAND_MAX,
                -5+rand()*10./RAND_MAX,
                rand()*(m-1)*1./RAND_MAX,
                rand()*(n-1)*1./RAND_MAX,
                -5+rand()*10./RAND_MAX};
    double x,y,z;
    int k=yBilinear::Line(m,n,L,A,B);
    bool b=0;
    for(int j=0,q;j<k;++j)
      if((q=yBilinear::CellIntersect(Z,A[j],B[j],L,x,y,z))>2){
        b=!(q-4);break;}
    if(b){
      int j=int(x/(m-1)*M),k=int(y/(n-1)*N);
      if(j>=0&&j<M&&k>=0&&k<N)Rainbow(yBmp::GetPixel(I,j,k),z,0,1);}}
  yBmp::WriteBmpFile("intersect.bmp",I);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

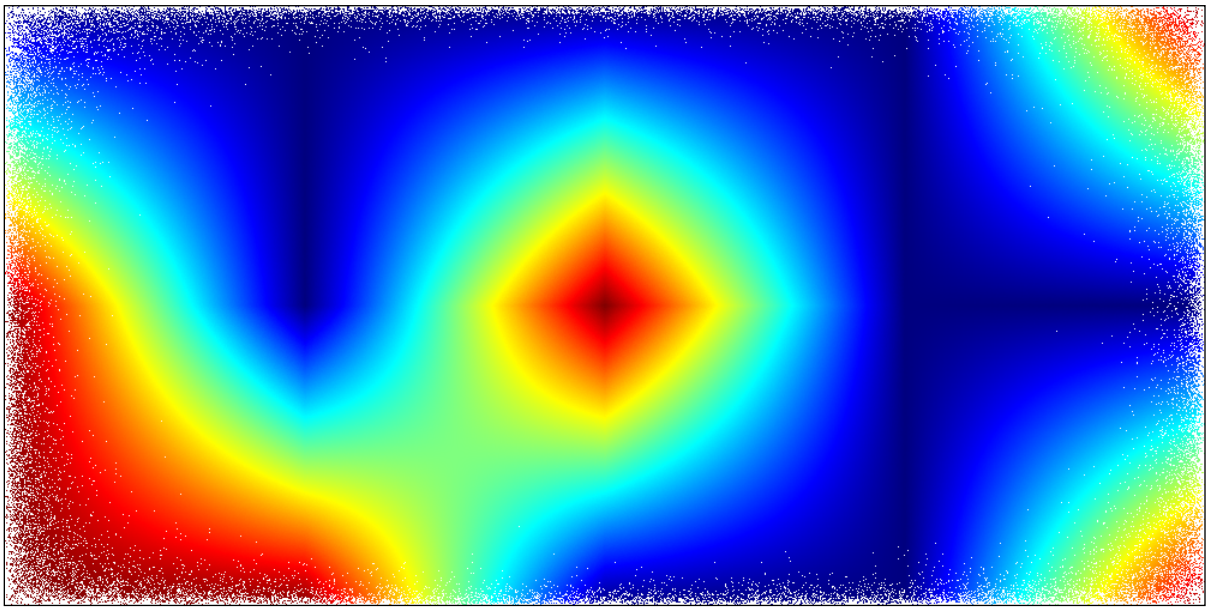


Fig. 9 Image generated by example code from Section 6.5.

Note that the bottom-left corner of the image is similar to the image that is displayed in Fig. 8. However, the image from Fig. 8 has fewer missing pixels than the image from Fig. 9. This is because different intersection criteria were used to generate the 2 images; only line-segment intersections were plotted in Fig. 9.

7. Calculating Surface Gradients: The Gradient() Function

The Gradient() function uses Eq. 39 to calculate the surface gradient at a user-specified location. To simplify the Gradient() function, Eq. 39 has been rewritten in terms of scaled coordinates using the substitutions from Eq. 44:

$$(\nabla\phi)_{s_x} = (s_y - j - 1)(z_{i,j} - z_{i+1,j}) - (s_y - j)(z_{i,j+1} - z_{i+1,j+1}). \tag{58}$$

$$(\nabla\phi)_{s_y} = (s_x - i - 1)(z_{i,j} - z_{i,j+1}) - (s_x - i)(z_{i+1,j} - z_{i+1,j+1}). \tag{59}$$

Eq. 60 can be used to find $(\nabla\phi)_x$ and $(\nabla\phi)_y$:

$$(\nabla\phi)_x = \frac{(\nabla\phi)_{s_x}}{\Delta x} \text{ and } (\nabla\phi)_y = \frac{(\nabla\phi)_{s_y}}{\Delta y}. \tag{60}$$

The components of the gradient can be used to determine the direction of steepest ascent, the slope in the direction of steepest ascent (Eq. 40), and a unit vector that is normal to the surface (Eq. 41).

7.1 Gradient() Code

```
template<class T>void Gradient(//<----------------------------SURFACE GRADIENT
    const T&Z,//<----------------------POINTER TO Z WHERE Z[i][j]=Z(X[i],Y[j])
    int i,int j,//<-------------------------GRID INDICES (0<=i<m-1 & 0<=j<n-1)
    double sx,double sy,//<--------------A GRID LOCATION IN SCALED COORDINATES
    double&delsx,double&delsy){//<-----SCALED GRADIENT COMPONENTS (CALCULATED)
  delsx=(sy-j-1)*(Z[i][j]-Z[i+1][j])-(sy-j)*(Z[i][j+1]-Z[i+1][j+1]);
  delsy=(sx-i-1)*(Z[i][j]-Z[i][j+1])-(sx-i)*(Z[i+1][j]-Z[i+1][j+1]);
}//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

7.2 Gradient() Parameters

Z — **Z** points to a 2-index array of values that represents $z(x, y)$, where **Z**[**i**][**j**] = $z(x_i, y_j)$.

i — **i** specifies i, an index that is associated with the x axis. Values for **i** cannot be less than zero or greater than $m - 2$, where m is the first-index size of **Z**.

j — **j** specifies j, an index that is associated with the y axis. Values for **j** cannot be less than zero or greater than $n - 2$, where n is the second-index size of **Z**.

sx — **sx** specifies s_x, a scaled distance from Eq. 42.

sy — **sy** specifies s_y, a scaled distance from Eq. 42.

delsx **delsx** is the calculated value for $(\nabla\phi)_{s_x}$, the *x*-component of the surface gradient (in scaled coordinates) from Eq. 58.

delsy **delsy** is the calculated value for $(\nabla\phi)_{s_y}$, the *y*-component of the surface gradient (in scaled coordinates) from Eq. 59.

7.3 Gradient() Simple Example

The following example begins by defining the surface that is shown in Fig. 5. The Gradient() function is then used to estimate the slope of the surface at $x=1.5$ and $y=0.5$.

```
#include <cstdio>//.....................................................printf()
#include "y_bilinear.h"//.............................yBilinear,<cmath>{sqrt()}
int main(){//<====================A SIMPLE EXAMPLE USING THE Gradient() FUNCTION
  const int m=5,n=3;
  double Z[m][n]={1,1,0 , 1,0,0 , 0,1,0 , 0,0,0 , 1,0,1};
  double x=1.5,y=.5;
  int i=yBilinear::SafeIndex(m,x),j=yBilinear::SafeIndex(n,y);
  double delx,dely;/*<-*/yBilinear::Gradient(Z,i,j,x,y,delx,dely);
  printf("At x=%.1f and y=%.1f, slope=%.1f\n",x,y,sqrt(delx*delx+dely*dely));
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

OUTPUT:

```
At x=1.5 and y=0.5, slope=0.0
```

7.4 Gradient() Image Example

The following example begins by defining the surface that is shown in Fig. 5. Next, functions from the yBmp namespace are used, along with the Gradient() function and the Rainbow() function (presented in Section 4.5), to create a pseudo-color image of the gradient of the surface. Finally, the image is written to a BMP file, which is displayed in Fig. 10. The color-to-value key is shown in Fig. 7.

Note that at the boundaries between cells, the gradient may contain discontinuities. The discontinuities are the result of the manner in which the surface has been constructed.

```
#include "y_bmp.h"//.....................................................yBmp
#include "y_bilinear.h"//.......................yBilinear,<cmath>{fabs(),sqrt()}
int main(){//<====================CREATE AN IMAGE FROM A SURFACE USING Gradient()
  const int m=5,n=3,M=1000,N=500;
  double Z[m][n]={1,1,0 , 1,0,0 , 0,1,0 , 0,0,0 , 1,0,1};
  unsigned char*I=yBmp::NewImage(M,N,255);
  for(int p=0;p<M;++p)for(int q=0;q<N;++q){
    double x=p*(m-1)*1./(M-1),y=q*(n-1)*1./(N-1);
    int i=yBilinear::SafeIndex(m,x),j=yBilinear::SafeIndex(n,y);
    double delx,dely;/*<-*/yBilinear::Gradient(Z,i,j,x,y,delx,dely);
    Rainbow(yBmp::GetPixel(I,p,q),sqrt(delx*delx+dely*dely),0,1.5);}
  yBmp::WriteBmpFile("gradient.bmp",I);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

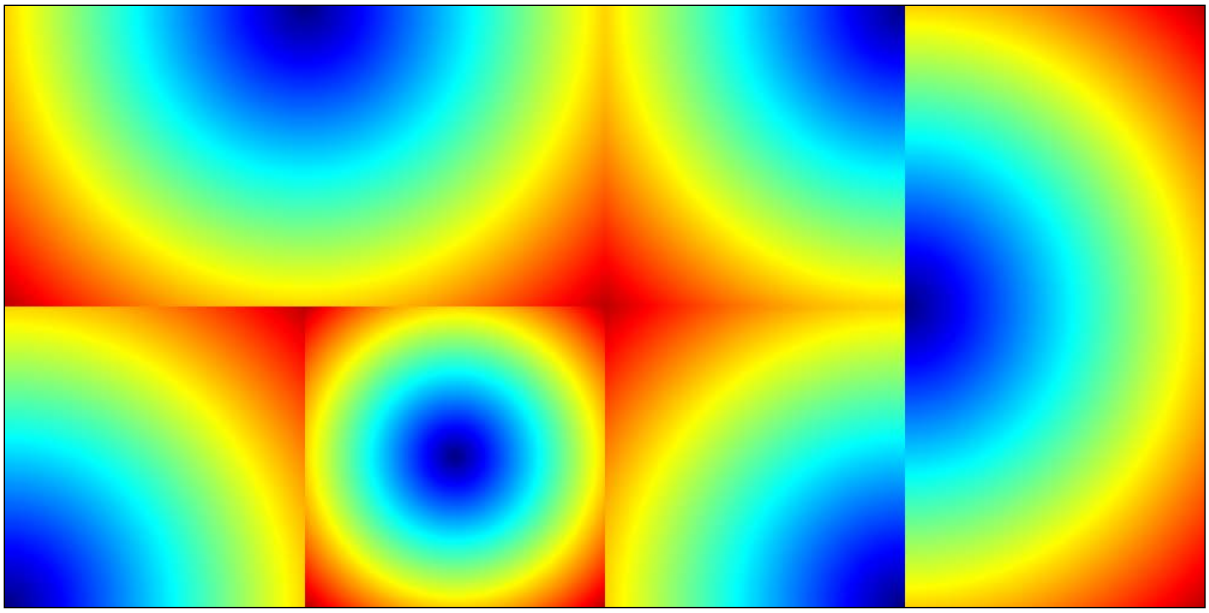


Fig. 10 Image generated by example code from Section 7.4.

8. Performance

The following example estimates the calculation time per 10^7 iterations for the Interpolate(), CellIntersect(), and Gradient() functions by randomly sampling and intersecting a randomly generated surface. The yRandom namespace is used to generate pseudorandom numbers. Figure 11 summarizes the results. Time values will vary based on computer specifications, compiler, compiler settings, etc.

```
#include <cstdio>//.....................................................printf()
#include <ctime>//....................................clock(),CLOCKS_PER_SEC
#include "y_bilinear.h"//.............................................yBilinear
#include "y_random.h"//................................................yRandom
int main(){//<=TEST THE SPEEDS OF Interpolate(), CellIntersect(), and Gradient()
  const int n=14,N=1<<n,M=10000000;
  unsigned I[625];/*<-*/yRandom::Initialize(I,1);//....state of Mersenne twister
  double**Z=new double*[N];/*<-*/for(int i=0;i<N;++i)Z[i]=new double[N];
  for(int i=0;i<N;++i)for(int j=0;j<N;++j)Z[i][j]=yRandom::RandU(I,0,1);
  printf("    size  |   Interpolation()   |   CellIntersect()   |     Gradient("
    ")\n     of   |---------------------|---------------------|---------------"
    "-----\n   array  |          |    z     |          | intersect|          | "
    "  delx\n    (m)   | time (s) |   avg.   | time (s) |   avg.   | time (s) |"
    "   avg.\n ---------|----------|----------|----------|----------|----------"
    "|----------\n");//.............................................table header
  for(int m=2;m<=N;m*=2){
    double s=0,t=clock();//.........................begin test for Interpolate()
    for(int k=0;k<M;++k){
      double x=yRandom::RandU(I,0,m-1),y=yRandom::RandU(I,0,m-1);
```

```
      int i=yBilinear::SafeIndex(m,x),j=yBilinear::SafeIndex(m,y);
      double z=yBilinear::Interpolate(Z,i,j,x,y);
      s+=z;}
    printf("%7d   |%8.3f  |%8.3f  |",m,(clock()-t)/CLOCKS_PER_SEC,s/M);
    s=0,t=clock();//.............................begin test for CellIntersect()
    for(int k=0;k<M;++k){
      double L[6]={yRandom::RandU(I,0,m-1),yRandom::RandU(I,0,m-1),
        yRandom::RandU(I,-1,1),yRandom::RandU(I,0,m-1),
        yRandom::RandU(I,0,m-1),yRandom::RandU(I,-1,1)};
      double x,y,z;
      int i=yRandom::RandI(I,0,m-2),j=yRandom::RandI(I,0,m-2);
      int test=yBilinear::CellIntersect(Z,i,j,L,x,y,z);
      if(test==4)s++;}
    printf("%9.3f |%8.3f  |",(clock()-t)/CLOCKS_PER_SEC,s/M);
    s=0,t=clock();//.................................begin test for Gradient()
    for(int k=0;k<M;++k){
      double x=yRandom::RandU(I,0,m-1),y=yRandom::RandU(I,0,m-1),delx,dely;
      int i=yBilinear::SafeIndex(m,x),j=yBilinear::SafeIndex(m,y);
      yBilinear::Gradient(Z,i,j,x,y,delx,dely);
      s+=delx;}
    printf("%9.3f |%8.3f\n",(clock()-t)/CLOCKS_PER_SEC,s/M);}
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

OUTPUT:

size of array (m)	Interpolation() time (s)	Interpolation() z avg.	CellIntersect() time (s)	CellIntersect() intersect avg.	Gradient() time (s)	Gradient() delx avg.
2	0.234	0.672	1.201	0.275	0.234	0.071
4	0.297	0.439	1.216	0.055	0.297	0.201
8	0.296	0.439	1.248	0.014	0.297	-0.020
16	0.280	0.486	1.233	0.005	0.265	-0.008
32	0.281	0.494	1.232	0.002	0.281	-0.001
64	0.281	0.505	1.263	0.001	0.281	-0.000
128	0.297	0.504	1.357	0.001	0.296	-0.000
256	0.312	0.501	1.420	0.000	0.296	0.000
512	0.328	0.500	1.404	0.000	0.296	-0.000
1024	0.484	0.500	1.591	0.000	0.452	-0.000
2048	0.905	0.500	2.028	0.000	0.874	0.000
4096	1.014	0.500	2.106	0.000	1.045	0.000
8192	1.154	0.500	2.372	0.000	1.076	-0.000
16384	1.451	0.500	2.558	0.000	1.311	-0.000

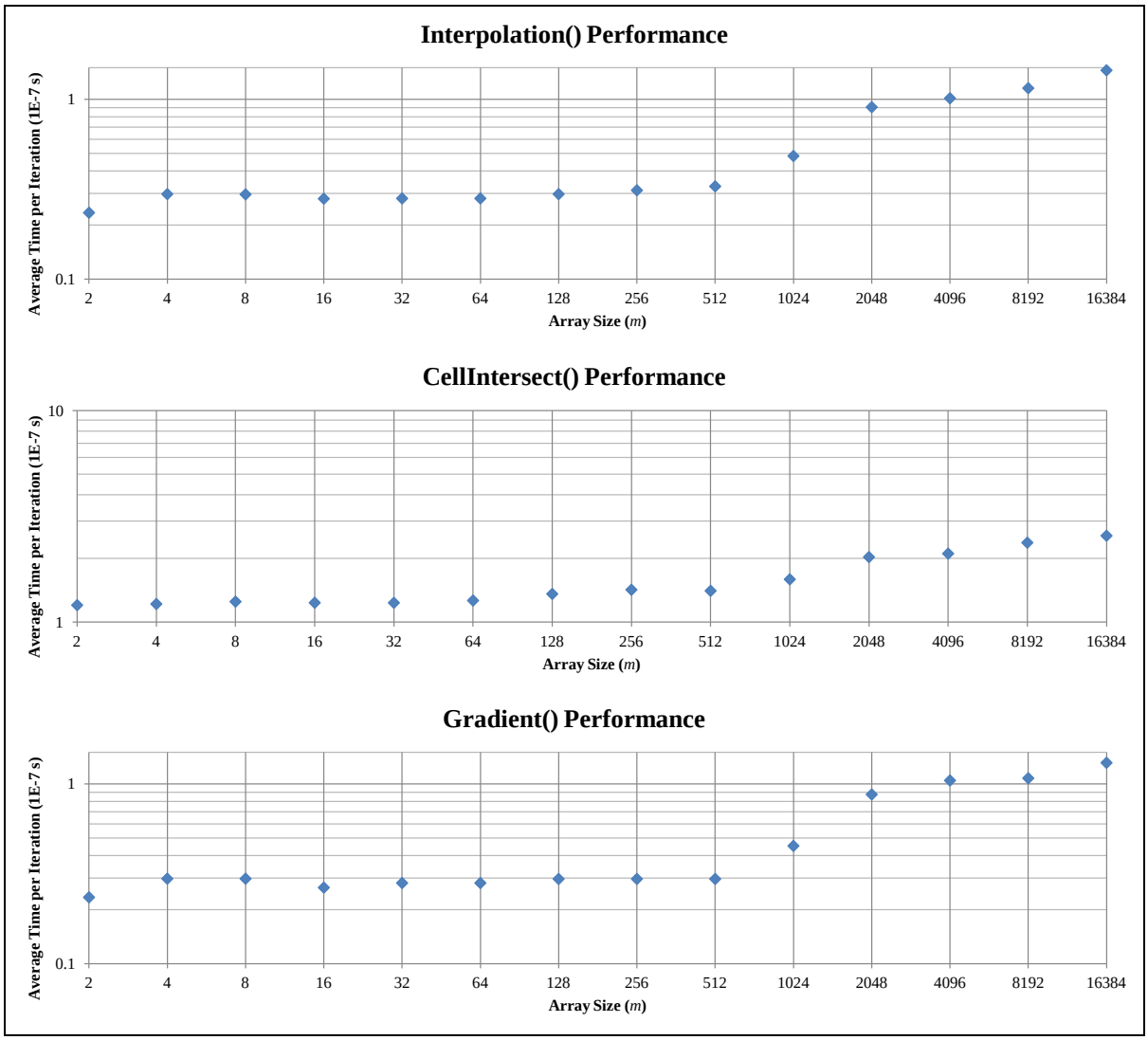


Fig. 11 Average time per iteration for the Interpolate(), CellIntersect(), and Gradient() functions

9. Performance II

The following example begins by defining 2 functions that can be used to find intersections between lines and evenly spaced rectangular surface grids. The first function, SimpleIntersect(), uses the CellIntersect() function to check all of the grid's cells. The second function, EfficientIntersect(), uses the CellIntersect() function to check a subset of the grid's cells, as specified by the Line() function.

Next, the example code estimates the calculation time per 10^5 iterations for the SimpleIntersect() and EfficientIntersect() functions. The yRandom namespace is used to generate pseudorandom numbers.

The example demonstrates 2 things: First, that using the Line() function to reduce the number of cells to check for intersection can result in large increases in performance. Second, average intersect values match between the 2 methods, providing evidence that the algorithm for the Line() function is valid.

```
template<class T>bool SimpleIntersect(//<=================LINE-GRID INTERSECTION
    const T&Z,//<------------------------POINTER TO Z WHERE Z[i][j]=Z(X[i],Y[j])
    int m,int n,//<-----------------------------------------------------GRID SIZES
    const double L[6],//<---------------A LINE SEGMENT {L0X,L0Y,L0Z,L1X,L1Y,L1Z}
    double&sx,double&sy,double&sz,//<---------POINT OF INTERSECTION (CALCULATED)
    double epsilon=1E-9){//<--------------------VALUE FOR DIVISION-BY-ZERO CHECK
  double d=-1,dt,xt,yt,zt;
  for(int i=0;i<m-1;++i)for(int j=0;j<n-1;++j)
    if(yBilinear::CellIntersect(Z,i,j,L,xt,yt,zt,epsilon)==4){
      xt-=L[0],yt-=L[1],zt-=L[2];
      if(dt=xt*xt+yt*yt+zt*zt<d||d<0)
        d=dt,sx=xt+L[0],sy=yt+L[1],sz=zt+L[2];}
  return d<0?0:1;
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

```
template<class T>bool EfficientIntersect(//<==============LINE-GRID INTERSECTION
    const T&Z,//<------------------------POINTER TO Z WHERE Z[i][j]=Z(X[i],Y[j])
    int m,int n,//<-----------------------------------------------------GRID SIZES
    const double L[6],//<---------------A LINE SEGMENT {L0X,L0Y,L0Z,L1X,L1Y,L1Z}
    double&sx,double&sy,double&sz,//<---------POINT OF INTERSECTION (CALCULATED)
    int*A,int*B,//<-----------STORAGE FOR CELL INDICES (FOR EACH, SIZE >= m+n-3)
    int k,//<-----------------------------RETURN VALUE FROM THE Line() FUNCTION
    double epsilon=1E-9){//<--------------------VALUE FOR DIVISION-BY-ZERO CHECK
  int q=0;/*<-*/for(int j=0;j<k;++j)
    if((q=yBilinear::CellIntersect(Z,A[j],B[j],L,sx,sy,sz,epsilon))>2)break;
  return !(q-4);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

```
#include <cstdio>//.....................................................printf()
#include <ctime>//.....................................clock(),CLOCKS_PER_SEC
#include "y_bilinear.h"//.............................................yBilinear
#include "y_random.h"//...............................................yRandom
int main(){//<========COMPARE Line() FUNCTION PERFORMANCE TO AN EXAUSTIVE METHOD
  const int n=11,N=1<<n,M=100000;
  unsigned I[625];/*<-*/yRandom::Initialize(I,1);//....state of Mersenne twister
  double**Z=new double*[N];/*<-*/for(int i=0;i<N;++i)Z[i]=new double[N];
  for(int i=0;i<N;++i)for(int j=0;j<N;++j)Z[i][j]=yRandom::RandU(I,0,1);
  printf("    size  |   SimpleIntersect()  | EfficientIntersect()\n"
         "     of   |----------------------|----------------------\n"
         "   array  |          | intersect |          | intersect\n"
         "    (m)   | time (s) |    avg.   | time (s) |   avg.\n"
         " ---------|----------|-----------|----------|-----------\n");
  for(int m=2;m<=N;m*=2){
    double s,t;
    s=0,t=clock(),yRandom::Initialize(I,1);
    for(int k=0;k<M;++k){
```

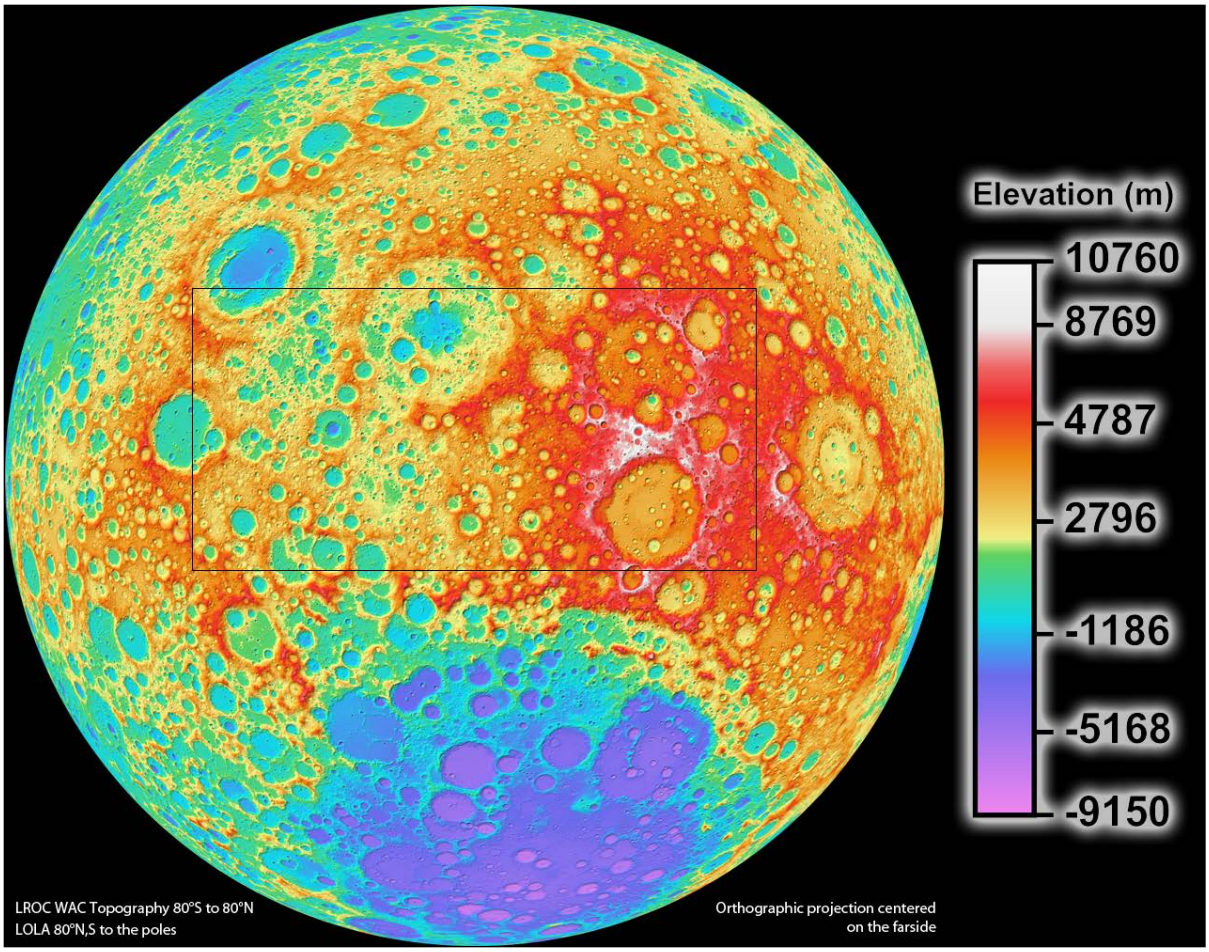
```
      double L[6]={yRandom::RandU(I,0,m-1),yRandom::RandU(I,0,m-1),
        yRandom::RandU(I,-1,1),yRandom::RandU(I,0,m-1),
        yRandom::RandU(I,0,m-1),yRandom::RandU(I,-1,1)};
      double x,y,z;
      int i=yRandom::RandI(I,0,m-2),j=yRandom::RandI(I,0,m-2);
      bool test=SimpleIntersect(Z,m,m,L,x,y,z);
      if(test)s++;}
    printf("%7d   |%9.3f |%10.7f |",m,(clock()-t)/CLOCKS_PER_SEC,s/M);
    s=0,t=clock(),yRandom::Initialize(I,1);
    int *A=new int[m+m-3],*B=new int[m+m-3];
    for(int k=0;k<M;++k){
      double L[6]={yRandom::RandU(I,0,m-1),yRandom::RandU(I,0,m-1),
        yRandom::RandU(I,-1,1),yRandom::RandU(I,0,m-1),
        yRandom::RandU(I,0,m-1),yRandom::RandU(I,-1,1)};
      int K=yBilinear::Line(m,m,L,A,B);
      double x,y,z;
      int i=yRandom::RandI(I,0,m-2),j=yRandom::RandI(I,0,m-2);
      bool test=EfficientIntersect(Z,m,m,L,x,y,z,A,B,K);
      if(test)s++;}
    delete[]A,delete[]B;
    printf("%9.3f |%10.7f\n",(clock()-t)/CLOCKS_PER_SEC,s/M);}
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

OUTPUT:

```
    size   |   SimpleIntersect()   | EfficientIntersect()
     of    |-----------------------|----------------------
   array   |           | intersect |           | intersect
    (m)    | time (s)  |    avg.   | time (s)  |   avg.
 ----------|-----------|-----------|-----------|-----------
      2    |     0.015 | 0.2961400 |     0.015 | 0.2961400
      4    |     0.047 | 0.2324000 |     0.016 | 0.2324000
      8    |     0.219 | 0.4265600 |     0.047 | 0.4265600
     16    |     1.061 | 0.5226000 |     0.046 | 0.5226000
     32    |     4.166 | 0.5722800 |     0.078 | 0.5722800
     64    |    17.044 | 0.6091200 |     0.110 | 0.6091200
    128    |    67.985 | 0.6480700 |     0.218 | 0.6480700
    256    |   272.998 | 0.6754100 |     0.406 | 0.6754100
    512    |  1089.319 | 0.6941000 |     0.748 | 0.6941000
   1024    |  4402.322 | 0.7071800 |     1.498 | 0.7071800
   2048    |17427.493  | 0.7174700 |     3.822 | 0.7174700
```

10. Surface-Elevation Example: The Far Side of the Moon

The image shown in Fig. 12 was obtained from NASA's Website. It presents a topographic view of the far side of the moon. Prior to working with the image, the file was converted from JPG format to BMP format using Microsoft Paint.



Reprinted from NASA's Goddard Space Flight Center/DLR/ASU. LRO Camera Team Releases High Resolution Global Topographic Map of Moon, 16 Nov 2011. [accessed 2014 May 29]. http://www.nasa.gov/mission_pages/LRO/news/lro-topo.html.

Fig. 12 Topographic image of the far side of the moon

The example code that follows begins by using the yBmp namespace to read the image file that is presented in Fig. 12 into memory. Next, the code uses a vertical line of pixels to interpret the scale that is shown to the right of the moon. The vertical line is placed horizontally in the center of the scale, with the starting location at the bottom-most position in the scale and the ending location at the top-most position.

Next, the code calculates elevation values for a portion, outlined in black, of the image of the moon in Fig. 12. This is accomplished by reading pixel values from the image and performing nearest-neighbor searches between the pixel color values and a path in 3-dimensional Cartesian color space that is defined by the values that have been read from the scale. Although the scale in figure 12 isn't linear, for this simple example, it's treated as if it is.

Finally, the code creates 3 images based on the elevation data obtained from Fig. 12.

```
#include "y_bmp.h"//.....................................................yBmp
#include "y_bilinear.h"//.....................yBilinear,<cmath>{fabs(),sqrt()}
int main(){//<==============CREATE VARIOUS IMAGES BASED ON MOON TOPOGRAPHIC DATA
  //------------------READ ORIGINAL IMAGE AND CAPTURE SCALE---------------------
  unsigned char*I=yBmp::ReadBmpFile("604359main_WAC_CSHADE_O000N1800_1000.bmp");
  double*X=new double[582];//.............................scale values [0,1]
  unsigned char**C=new unsigned char*[582];//.scale colors ({B,G,R} and [0,255])
  for(int k=0;k<582;++k)X[k]=k/581.,C[k]=yBmp::GetPixel(I,1060,k+143);
  //----------CAPTURE AND INTERPRET A PORTION OF THE ORIGINAL IMAGE-------------
  double Z[600][300];//.....................................elevation values
  for(int p=0;p<600;++p)for(int q=0;q<300;++q){
    unsigned char*c=yBmp::GetPixel(I,p+200,q+400);
    double d,D=-1;
    for(int k=0;k<582;++k){
      d=(C[k][0]-c[0])*(C[k][0]-c[0])+(C[k][1]-c[1])*(C[k][1]-c[1])
        +(C[k][2]-c[2])*(C[k][2]-c[2]);
      if(d<D||D<0)D=d,Z[p][q]=X[k];}}
  //--------CREATE A NEW (MONOCHROME) IMAGE BASED ON ELEVATION VALUES-----------
  unsigned char*I2=yBmp::NewImage(600,300,255);
  for(int p=0;p<600;++p)for(int q=0;q<300;++q){
    unsigned char*c=yBmp::GetPixel(I2,p,q);
    c[0]=c[1]=c[2]=int(Z[p][q]*255);}
  yBmp::WriteBmpFile("moon_topography.bmp",I2);
  //-------------------CREATE A SURFACE-OCCULTATION IMAGE----------------------
  double L[6]={0,0,5};//.....................................vantage point
  int*A=new int[600+300-3],*B=new int[600+300-3];
  for(int p=0;p<600;++p)for(int q=0;q<300;++q){
    L[3]=p,L[4]=q,L[5]=Z[p][q]+.00000001;//.....................viewable point?
    int K=yBilinear::Line(600,300,L,A,B),Q=0;
    double x,y,z;
    for(int k=0;k<K&&Q!=4;++k)Q=yBilinear::CellIntersect(Z,A[k],B[k],L,x,y,z);
    if(Q==4){
      unsigned char*c=yBmp::GetPixel(I2,p,q);
      c[0]=c[1]=c[2]=0;}}
  yBmp::WriteBmpFile("moon_occultation.bmp",I2);
  //----------------------CREATE SURFACE-GRADIENT IMAGE-----------------------
  for(int p=0;p<600;++p)for(int q=0;q<300;++q){
    int i=yBilinear::SafeIndex(600,p),j=yBilinear::SafeIndex(300,q);
    double delx,dely;/*<-*/yBilinear::Gradient(Z,i,j,p,q,delx,dely);
    Rainbow(yBmp::GetPixel(I2,p,q),sqrt(delx*delx+dely*dely),0,.4);}
  yBmp::WriteBmpFile("moon_gradient.bmp",I2);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~
```

Figure 13 presents a topographic view of the moon with elevations denoted by shades of gray.

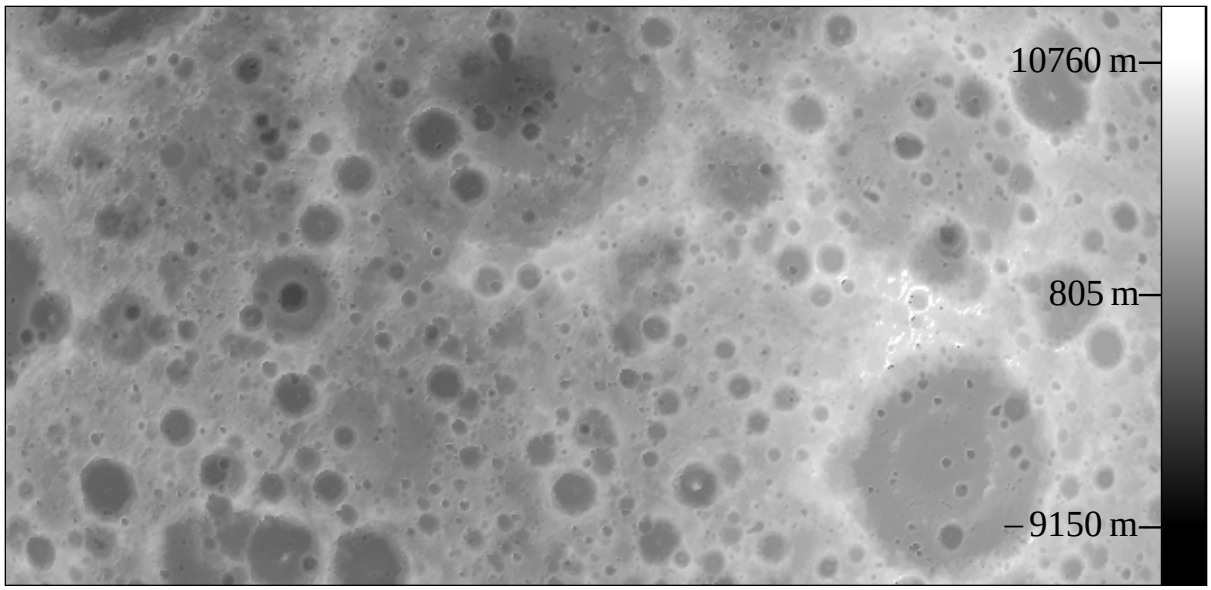


Fig. 13 Topographic view of a portion of the far side of the moon

Figure 14 was created by placing a vantage point at the lower-left corner of the image and at an elevation of 90,400 m. The Line() and CellIntersect() functions were used to determine line of sight. Any point that was not visible from the vantage point was recolored to black.

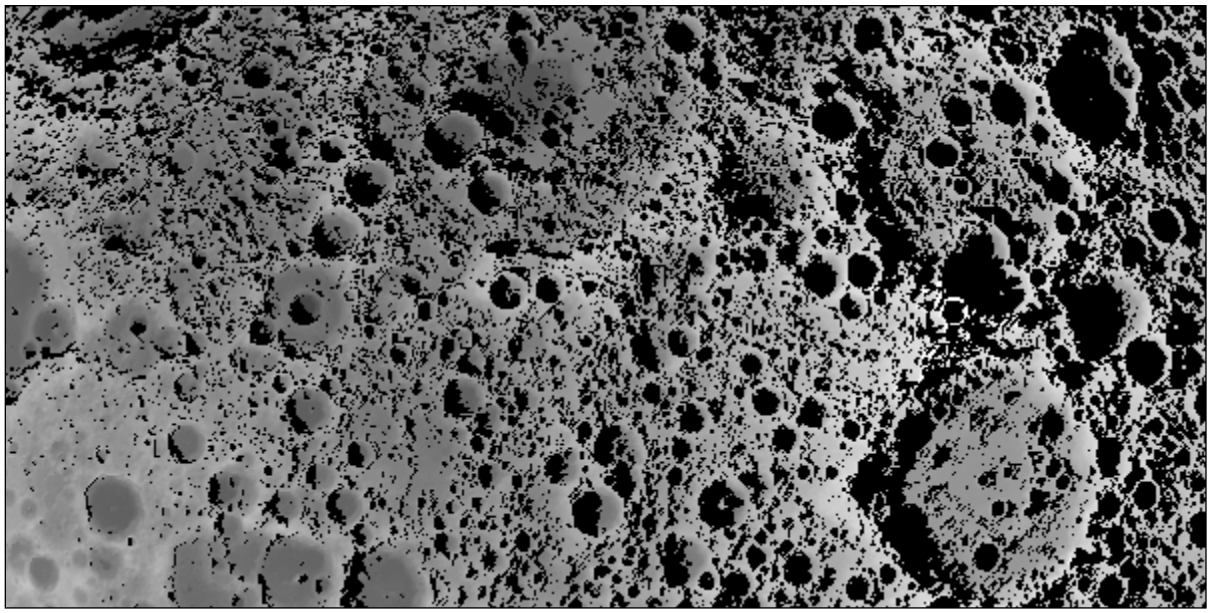


Fig. 14 Topographic view of the moon but with surface occultations drawn in black

Figure 15 shows the magnitude of the scaled surface gradient for the topographic information that is shown in Fig. 13.

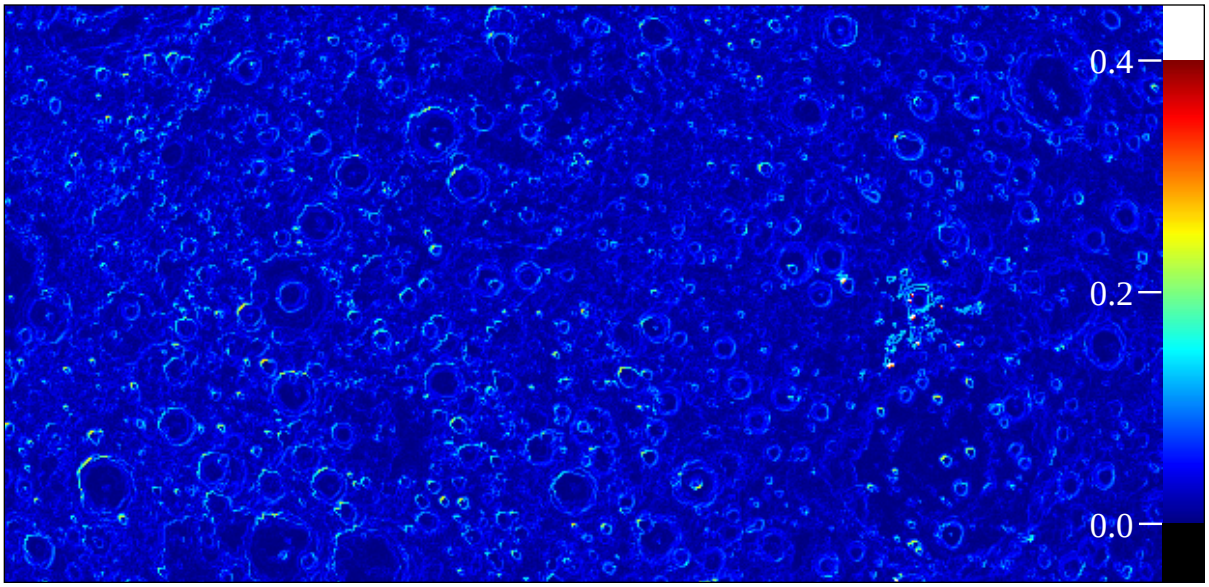


Fig. 15 View of the magnitude of the scaled surface gradient for a portion of the far side of the moon

11. Code Summary

A summary sheet is provided at the end of this report. It presents the yBilinear namespace, which contains the SafeIndices(), Interpolate(), CellIntersect(), Line(), and Gradient() functions.

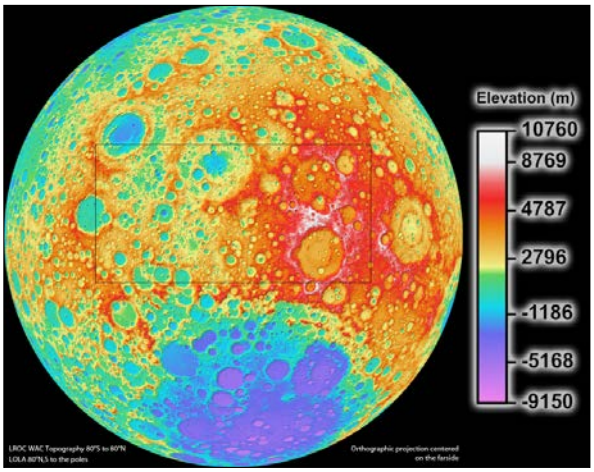
y_bilinear.h

```
#ifndef Y_BILINEAR_GUARD//          See Yager, R.J. "Working with Evenly Spaced,
#define Y_BILINEAR_GUARD//                Rectangular Surface Grids Using C++"
#include <cmath>//.....................................................fabs(),sqrt()
#include <cstdlib>//....................................................abs()
namespace yBilinear{//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  inline int SafeIndex(//<=====CALCULATE A SAFE INDEX AT A GIVEN SCALED LOCATION
      int m,//<------------------NUMBER OF GRID INDICES (m SHOULD BE AT LEAST 2)
      double s){//<-------------------------A GRID LOCATION IN SCALED COORDINATES
    return s<0?0:s>m-2?m-2:int(s);
  }//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~~
  template<class T>double Interpolate(//<==================BILINEAR INTERPOLATION
      const T&Z,//<----------------------POINTER TO Z WHERE Z[i][j]=Z(X[i],Y[j])
      int i,int j,//<--------------------------GRID INDICES (0<=i<m-1 & 0<=j<n-1)
      double sx,double sy){//<-------------A GRID LOCATION IN SCALED COORDINATES
    double za=(sy-j)*(Z[i][j+1]-Z[i][j])+Z[i][j];
    return (sx-i)*((sy-j)*(Z[i+1][j+1]-Z[i+1][j])+Z[i+1][j]-za)+za;
  }//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~~
  template<class T>int CellIntersect(//<===================LINE-CELL INTERSECTION
      const T&Z,//<----------------------POINTER TO Z WHERE Z[i][j]=Z(X[i],Y[j])
      int i,int j,//<--------------------------GRID INDICES (0<=i<m-1 & 0<=j<n-1)
      const double L[6],//<----------A SCALED LINE {L0SX,L0SY,L0Z,L1SX,L1SY,L1Z}
      double&sx,double&sy,double&z,//<----SCALED INTERSECTION POINT (CALCULATED)
      double epsilon=1E-9){//<------------------VALUE FOR DIVISION-BY-ZERO CHECK
    double C0=L[3]-L[0],C1=L[4]-L[1],C2=L[5]-L[2];
    double C5=i-L[0],C6=j-L[1],C3=C5+1,C4=C6+1;
    double a=C0*C1*(Z[i][j]-Z[i+1][j]-Z[i][j+1]+Z[i+1][j+1]),
      b=-(C1*C3+C0*C4)*Z[i][j]+(C1*C5+C0*C4)*Z[i+1][j]
        +(C1*C3+C0*C6)*Z[i][j+1]-(C1*C5+C0*C6)*Z[i+1][j+1]-C2,
      c=C3*C4*Z[i][j]-C5*C4*Z[i+1][j]-C3*C6*Z[i][j+1]+C5*C6*Z[i+1][j+1]-L[2];
    double t,D;
    if(fabs(a)<epsilon&&fabs(b)<epsilon)return fabs(c)<epsilon?-1:0;
    if(fabs(a)<epsilon)t=-c/b,sx=C0*t+L[0],sy=C1*t+L[1],z=C2*t+L[2];
    else{
      if((D=b*b-4*a*c)<0)return 0;//..no intersection, <sx,sy,sz> not calculated
      t=(-b-sqrt(D))/2/a,sx=C0*t+L[0],sy=C1*t+L[1],z=C2*t+L[2];
      double tb=(-b+sqrt(D))/2/a,sxb=C0*tb+L[0],syb=C1*tb+L[1],zb=C2*tb+L[2];
      if(t>=0&&t<=1&&tb>=0&&tb<=1){
        bool ra=sx<i||sx>i+1||sy<j||sy>j+1?1:0;
        bool rb=sxb<i||sxb>i+1||syb<j||syb>j+1?1:0;
        if(ra&&rb&&t>tb){sx=sxb,sy=syb,z=zb;return 1;}
        else if(ra||!rb&&t>tb)t=tb,sx=sxb,sy=syb,z=zb;}
      else if(t>tb&&tb>0||t<=0&&t<tb)t=tb,sx=sxb,sy=syb,z=zb;}
    if(sx<i||sx>i+1||sy<j||sy>j+1)return 1;//.....................out of bounds
    return t<0?2:(t>1?3:4);//....line (2), ray (3), or segment (4) intersections
  }//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~~
  inline int Line(//<=======================================LINE-GRID INTERSECTION
      int m,int n,//<-----NUMBERS OF GRID INDICES (m AND n SHOULD BE AT LEAST 2)
      const double L[6],//<--A SCALED LINE SEGMENT {L0SX,L0SY,L0Z,L1SX,L1SY,L1Z}
      int*A,int*B){//<--------STORAGE FOR CELL INDICES (FOR EACH, SIZE >= m+n-3)
    int i=SafeIndex(m,*L),j=SafeIndex(n,L[1]),I=SafeIndex(m,L[3]),
      J=SafeIndex(n,L[4]),di=i<I?1:-1,dj=j<J?1:-1,bi=di>0?1:0,bj=dj>0?1:0,k=0;
    double M=J-j?(L[3]-*L)/(L[4]-L[1]):0,N=I-i?1/M:0,P=*L-L[1]*M,Q=-P*N;
    if(abs(I-i)>abs(J-j))for(int it;dj*(J-j)>=0;j+=dj,i-=di)//dj* selects < or >
      for(it=j-J?SafeIndex(m,(j+bj)*M+P):I;di*(i-it)<=0;i+=di)A[k]=i,B[k++]=j;
    else for(int jt;di*(I-i)>=0;i+=di,j-=dj)//.................di* selects < or >
      for(jt=i-I?SafeIndex(n,(i+bi)*N+Q):J;dj*(j-jt)<=0;j+=dj)A[k]=i,B[k++]=j;
    return k;//.........................the number of elements to check in A and B
  }//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~~
  template<class T>void Gradient(//<-----------------------------SURFACE GRADIENT
      const T&Z,//<----------------------POINTER TO Z WHERE Z[i][j]=Z(X[i],Y[j])
      int i,int j,//<--------------------------GRID INDICES (0<=i<m-1 & 0<=j<n-1)
      double sx,double sy,//<--------------A GRID LOCATION IN SCALED COORDINATES
      double&delsx,double&delsy){//<-----SCALED GRADIENT COMPONENTS (CALCULATED)
    delsx=(sy-j-1)*(Z[i][j]-Z[i+1][j])-(sy-j)*(Z[i][j+1]-Z[i+1][j+1]);
    delsy=(sx-i-1)*(Z[i][j]-Z[i][j+1])-(sx-i)*(Z[i+1][j]-Z[i+1][j+1]);
  }//~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~~
}//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#endif
```

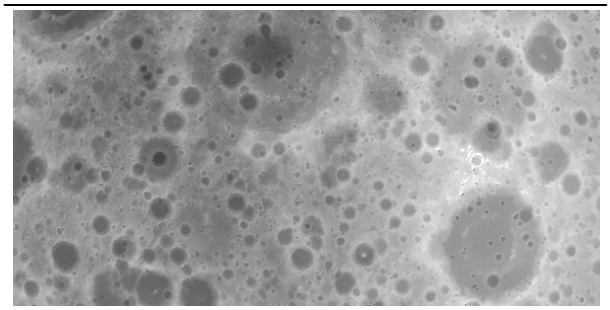
Topography Example - The Far Side of the Moon

```
#include "y_bmp.h"//.....................................................yBmp
#include "y_bilinear.h"//.....................yBilinear,<cmath>{fabs(),sqrt()}
int main(){//<===============CREATE VARIOUS IMAGES BASED ON MOON TOPOGRAPHIC DATA
  //------------------READ ORIGINAL IMAGE AND CAPTURE SCALE----------------------
  unsigned char*I=yBmp::ReadBmpFile("604359main_WAC_CSHADE_O000N1800_1000.bmp");
  double*X=new double[582];//.............................scale values [0,1]
  unsigned char**C=new unsigned char*[582];//.scale colors ({B,G,R} and [0,255])
  for(int k=0;k<582;++k)X[k]=k/581.,C[k]=yBmp::GetPixel(I,1060,k+143);
  //----------CAPTURE AND INTERPRET A PORTION OF THE ORIGINAL IMAGE-------------
  double Z[600][300];//.....................................elevation values
  for(int p=0;p<600;++p)for(int q=0;q<300;++q){
    unsigned char*c=yBmp::GetPixel(I,p+200,q+400);
    double d,D=-1;
    for(int k=0;k<582;++k){
      d=(C[k][0]-c[0])*(C[k][0]-c[0])+(C[k][1]-c[1])*(C[k][1]-c[1])
        +(C[k][2]-c[2])*(C[k][2]-c[2]);
      if(d<D||D<0)D=d,Z[p][q]=X[k];}}
  //--------CREATE A NEW (MONOCHROME) IMAGE BASED ON ELEVATION VALUES-----------
  unsigned char*I2=yBmp::NewImage(600,300,255);
  for(int p=0;p<600;++p)for(int q=0;q<300;++q){
    unsigned char*c=yBmp::GetPixel(I2,p,q);
    c[0]=c[1]=c[2]=int(Z[p][q]*255);}
  yBmp::WriteBmpFile("moon_topography.bmp",I2);
  //-------------------CREATE A SURFACE-OCCULTATION IMAGE----------------------
  double L[6]={0,0,5};//.........................................vantage point
  int*A=new int[600+300-3],*B=new int[600+300-3];
  for(int p=0;p<600;++p)for(int q=0;q<300;++q){
    L[3]=p,L[4]=q,L[5]=Z[p][q]+.00000001;//.....................viewable point?
    int K=yBilinear::Line(600,300,L,A,B),Q=0;
    double x,y,z;
    for(int k=0;k<K&&Q!=4;++k)Q=yBilinear::CellIntersect(Z,A[k],B[k],L,x,y,z);
    if(Q==4){
      unsigned char*c=yBmp::GetPixel(I2,p,q);
      c[0]=c[1]=c[2]=0;}}
  yBmp::WriteBmpFile("moon_occultation.bmp",I2);
  //----------------------CREATE SURFACE-GRADIENT IMAGE--------------------------
  for(int p=0;p<600;++p)for(int q=0;q<300;++q){
    int i=yBilinear::SafeIndex(600,p),j=yBilinear::SafeIndex(300,q);
    double delx,dely;/*<-*/yBilinear::Gradient(Z,i,j,p,q,delx,dely);
    Rainbow(yBmp::GetPixel(I2,p,q),sqrt(delx*delx+dely*dely),0,.4);}
  yBmp::WriteBmpFile("moon_gradient.bmp",I2);
}//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~16AUG2014~~~~~~~
```

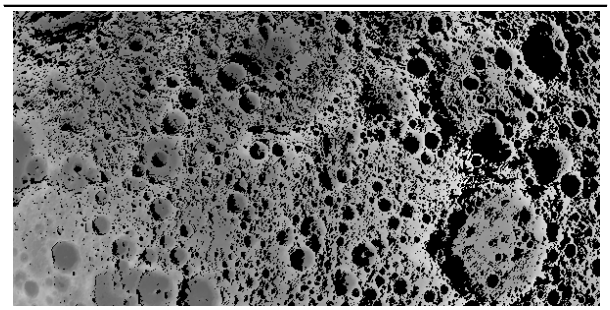
604359main_WAC_CSHADE_O000N1800_1000.bmp



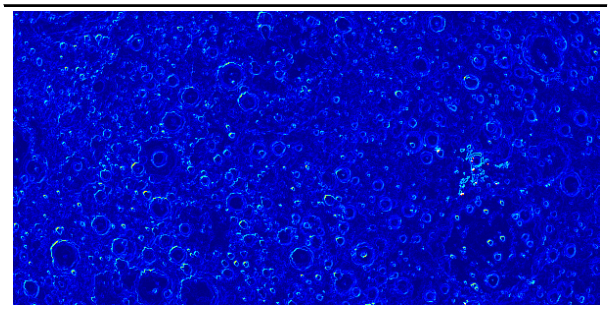
moon_topography.bmp



moon_occultation.bmp



moon_gradient.bmp



12. References

1. Yager RJ. Generating pseudorandom numbers from various distributions using C++. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2014 Jun. Report No.: ARL-TN-613.

2. Yager RJ. Reading, writing, and modifying BMP files using C++. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2013 Aug. Report No.: ARL-TN-559.

3. Boas ML. Mathematical methods in the physical sciences. 2nd ed. New York (NY): John Wiley & Sons; 1983. p. 229, 250, and 251.

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
1 (PDF)	DIR USARL RDRL WML A R YAGER

INTENTIONALLY LEFT BLANK.